SGM

Systèmes d'exploitation - Gestion de la mémoire

Master S.T.S. mention informatique, première année

Isabelle Puaut

Septembre 2011

Table des matières

Première partie

Généralités sur la gestion des informations

1 Modèle pour la désignation et la liaison

1.1 Terminologie

- *Objets logiques* : objets définis par l'utilisateur (variables, fichiers)
- *Objets physiques* : correspondance physique des objets logiques (secteur disque, emplacement mémoire)
- *Nom* : rôles d'un nom
  - identification d'un objet (permet de le distinguer des autres)
  - permettre l'accès à l'objet (le retrouver et le manipuler)
  - on parle généralement d'*identificateur* pour les objets logiques, d'*adresses* pour les objets physiques
- *Relation de désignation* : fait correspondre un objet à un nom
- *Liaison* : établissement de la correspondance entre nom et objet désigné

1.2 Système de désignation

Définition 1. *Un système de désignation permet de décrire les associations* (nom, objet désigné)*.*

Remarque 2. *Les associations* (nom, objet désigné) *peuvent varier au cours du temps : ajout/suppression de noms, changement d'objet désigné (par exemple les variables locales dans un programme)*

Eléments constitutifs d'un système de désignation :
- un *domaine de désignation* : l'ensemble des objets pouvant être désignés
- un *ensemble des noms* : l'ensemble des noms autorisés
- des *contextes de désignation* : une relation entre un ensemble de noms (lexique) et un ensemble d'objet
- un *réseau de désignation* : définit les liens entre les contextes.

Opérations sur les noms :
- *Lier* : lier un objet O à un nom N dans un contexte C
- *Résoudre* : résoudre un nom N dans un contexte C (chercher l'objet associé à N dans C)

Exemple : désignation dans le 8086

Exemple 3.
- *domaine de désignation* : ensemble des octets de la mémoire physique
- *ensemble des noms* : adresses relatives dans l'intervalle $[0..2^{16}-1]$
- *contexte de désignation* : si le programme est implanté à l'adresse 10000 (DS=10000), $0 \rightarrow 10000, 1 \rightarrow 10001$, etc.

Environnements de désignation limités et illimités

- Environnements *limités* (nombre limité *a priori* de noms). Il faut se préoccuper de la gestion des noms (allocation, libération) car au cours du temps le même nom sera utilisé à des fins différentes. Exemple : adresses mémoire.
- Environnements *illimités* (très grand nombre de noms). On n'a pas à se préoccuper de la gestion des noms. Exemples : noms externes de fichiers, adresses mémoire dans des processeurs 64-bits.

1.3 Résolution des noms

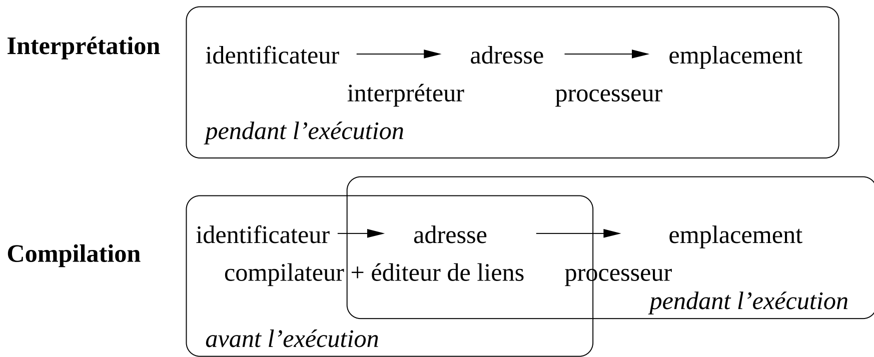
Résolution des noms : statique vs dynamique

Résolution d'un nom dans un contexte : retrouver l'objet associé au nom dans ce contexte. Types de résolution :

- Résolution *statique* : correspondance entre nom et objet effectuée une fois pour toute avant l'exécution
- Résolution *dynamique* : correspondance entre nom et objet re-calculée à chaque accès

Distinction bien connue pour traduction et exéc. de programmes :

- *Compilation* : remplacement des noms des objets logiques par les objets physiques correspondants (adresses) avant exécution
- *Interprétation* : on détermine lors de chaque accès à un objet logique l'objet physique correspondant

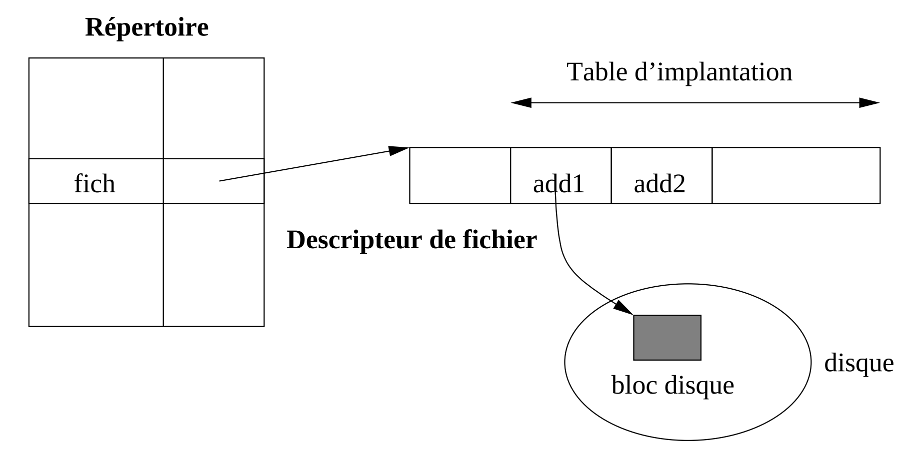


Exemple 4. **int** x ; ... x=0 ;

- Schéma compilé : on va décider, avant exécution d'implanter l'objet x à une certaine adresse (ex : 1020) et le programme exécutable contiendra une instruction dont l'opérande sera l'adresse 1020 (ex : mov ax,1020).
- Schéma interprété : l'affectation se fera via un appel de l'interpréteur de type *affecter(x,0)* qui déterminera l'emplacement associé à x, pendant l'exécution.

Résolution des noms : chaîne d'accès

- En général, passage de l'identificateur d'objet logique à l'objet physique est *indirect* (utilisation de noms ou objets intermédiaires)
- Résolution complète du nom : parcourir de cette chaîne d'accès en utilisant différents mécanismes
- Un *descripteur* est une structure de donnée de taille fixe qui contient à la fois des informations sur la localisation de l'objet (nom) et sur son "mode d'emploi" (protection, désignation des fonctions d'accès...)



Exemple 5 (Chaîne d'accès à un fichier).
- le passage du nom externe du fichier à l'adresse du descripteur est réalisé par l'algorithme de recherche dans le répertoire
- l'adresse du descripteur repère le descripteur
- l'adresse disque est obtenue par accès au contenu de la table d'implantation figurant dans le descripteur
- l'accès au bloc se fait à partir de l'adresse disque grâce au matériel de contrôle du disque.



1.4 Exemple : SGF

Exemple : système de gestion de fichiers (SGF)

- *Domaine de désignation* : fichiers et autres objets (sous UNIX, répertoires, fichiers spéciaux tels que périphériques caractère ou bloc, liens symboliques, tubes – pipes)
- *Ensemble des noms* : noms dont se sert l'utilisateur. On parle de nom *externes* pour les distinguer des noms utilisés par le système (*internes*). Deux types de noms sous UNIX :
  - noms simples (chaînes de caractères sans /)
  - chemins d'accès : nomsimple1/nomsimple2/... ou /nomsimple1/nomsimple2/ ...
- *contexte - environnement de désignation* : associe des noms simples à des objets. Un répertoire est un objet particulier qui représente un contexte, c'est un environnement de désignation. Il fait la correspondance entre nom externe et nom interne (i-node dans le cas d'UNIX).

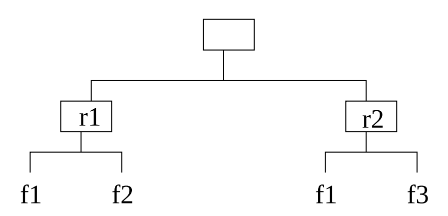
Répertoire

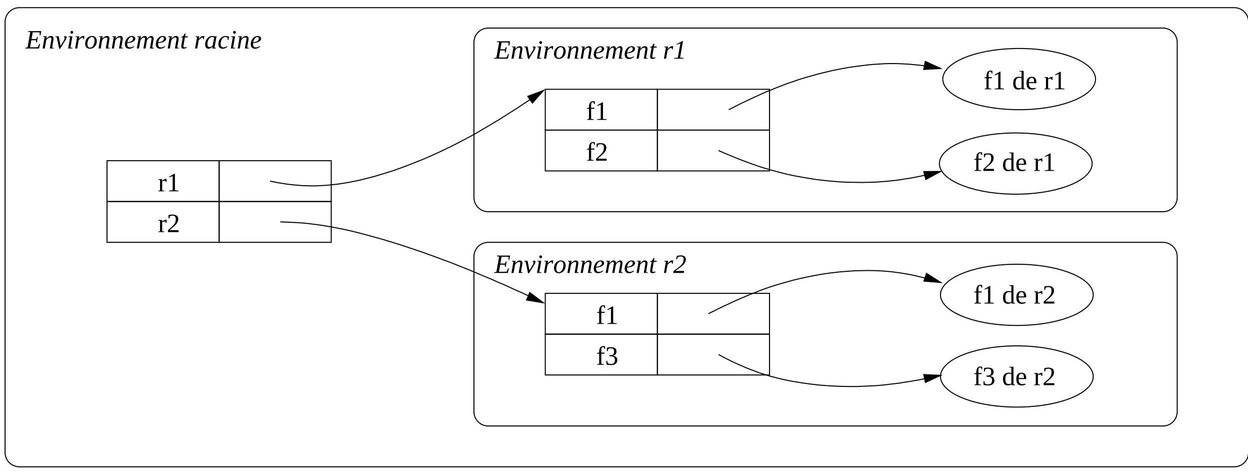
nom externe	nom interne
f1	
d1	

fichier f1

répertoire d1

- *Réseau de désignation* : la structure de la hiérarchie des fichiers constitue le réseau de désignation. Dans le cas d'une structure arborescente :
  - un répertoire peut contenir la description d'un fichier ou d'autres répertoires
  - répertoire particulier : *racine*. N'est décrit dans aucun autre répertoire





- *Règle d'interprétation des noms* :
  - un fichier peut être désigné sans ambiguité en donnant : le *nom simple* du fichier, le nom du répertoire le *contenant*
  - environnements utilisables sans les nommer explicitement : répertoire *racine* (pour les chemins d'accès absolus) et répertoire *de travail* (pour les chemins d'accès relatifs).
  - *règles de recherche* (PATH) : permettent de définir un ensemble de chemins d'accès à utiliser dans certaines conditions (PATH, LD_LIBRARY_PATH, CLASSPATH)
- *Liaison* : liaison à un contexte quand on intègre un fichier à un répertoire (création, copie, déplacement)

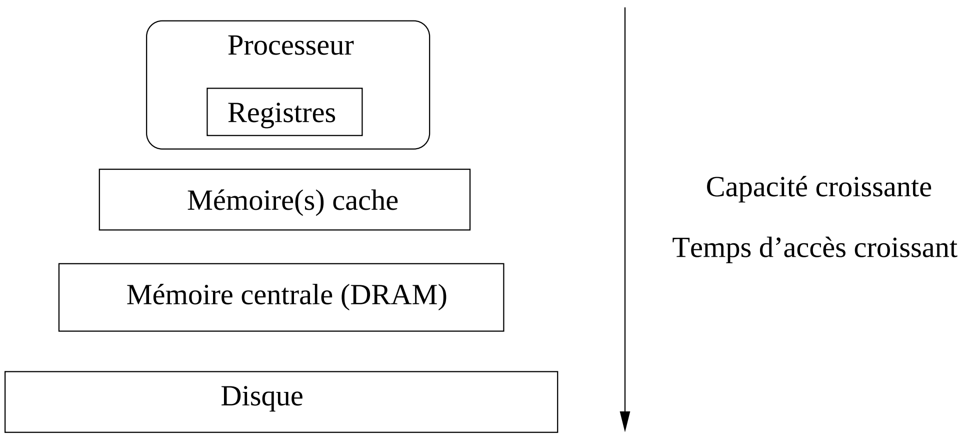
2 Hiérarchies de mémoire - caches

2.1 Notion de hiérarchie mémoire

- Supports divers de stockage d'information
- Capacité de stockage et temps d'accès très hétérogène
- En général, plus le temps d'accès à un support est rapide, plus sa capacité est faible

Definition 6 (Hiérarchie de mémoire)**.** Organisation des supports de stockage par temps d'accès croissants (ou taille croissante)

Notion de hiérarchie mémoire : exemple



- *Registres* : mémoire très rapide, directement accessible au processeur en un cycle, capacité de quelques dizaines à quelques centaines d'octets
- *Mémoire cache interne* (L1) : mémoire rapide, intégrée au processeur, accessible en quelques cycles, de capacité de quelques dizaines de Koctets (par exemple 16 Koctets)
- *Mémoires cache externes* (L2 - L3) : mémoires moins rapides que le cache L1 mais plus volumineux
- *Mémoire centrale* (Dynamic Random Access Memory) : capacité plusieurs Goctets, temps d'accès de l'ordre de plusieurs dizaines de cycles du processeur (100-300ns)
- *Disque* : capacité de plusieurs centaines de Goctets, temps d'accès de plusieurs dizaines de ms

Notion de hiérarchie mémoire : objectif

- *Objectif* : offrir à l'utilisateur l'espace de la mémoire la plus grande avec le temps d'accès de la mémoire la plus rapide
- *Principe* : on conserve à tout instant l'information la plus utilisée dans la mémoire la plus rapide
- *Pourquoi ça marche ?* : principe de localité

Definition 7 (Principe de localité)**.**

- *Localité spatiale* : si un élément est référencé à un instant donné, les emplacements voisins ont de fortes probabilités d'être référencés dans un futur proche (accès(a, t) $\implies$ probabilité forte d'accès(a+d, t+ϵ)
- *Localité temporelle* : un élément référencé à un instant a une forte probabilité d'être à nouveau référencé dans un futur proche (accès(a, t) $\implies$ probabilité forte d'accès(a, t+ϵ)

2.2 Principe général du mécanisme de cache

Principe général du mécanisme de cache - antémémoire

- *Origine du terme* : la mémoire rapide introduite entre le processeur et la mémoire proprement dit (caches matériels)
- *Dans les faits* : mécanisme général que l'on retrouve à plusieurs niveaux de la hiérarchie de mémoire (géré entièrement par matériel, ou par logiciel, ou conjointement par matériel et logiciel comme dans la pagination à la demande)
- Principes généraux de mise en œuvre existent, mais stratégies de mise en œuvre différentes selon le niveau

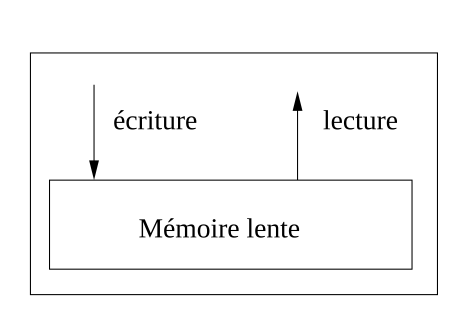
Soient deux niveaux contigus de la hiérarchie mémoire : mémoire *rapide* et mémoire *lente* :

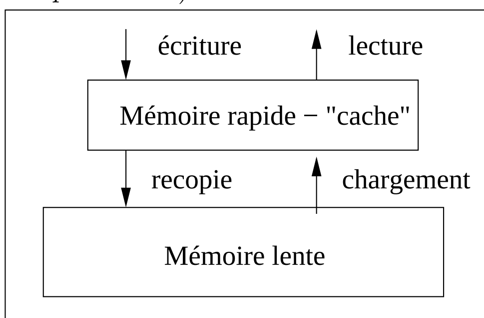
— *Adressage* : adressage de la mémoire lente, mais accès toujours réalisés sur la mémoire rapide

— *Accès* : si l'information voulue n'est pas présente dans la mémoire rapide il y a *défaut de cache*. Il faut alors la transférer de la mémoire lente vers la mémoire rapide (*chargement*)

— *Écriture* : écriture dans la mémoire rapide, recopie de l'information en mémoire lente, éventuellement de manière différée (*recopie*)

— Défaut de cache quand la memoire rapide est pleine : enlever au préalable une information de la mémoire rapide (*remplacement*), en essayant de maintenir en mémoire rapide les informations les plus "utiles" (*politique de remplacement*).





Abstraction **Mise en oeuvre**

— la mémoire rapide joue le rôle de *cache* pour la mémoire lente : la mémoire rapide contient la partie "utile" de la mémoire lente

— L'efficacité du cache peut être évalué par le *taux de défaut* : le rapport entre le nombre d'accès provoquant un défaut et le nombre d'accès total (dépend fortement des programmes)

2.3 Eléments de mise en œuvre

— Quel que soit le niveau de la hiérarchie où on se place les problèmes à résoudre sont du même type, par contre les solutions apportées peuvent être assez différentes.

— Eléments à considérer :
  — Représentation de l'état du cache
  — Politique de recopie
  — Politique de remplacement

Mise en œuvre : représentation de l'état du cache

— Division du cache et de la mémoire lente en *blocs* de la même taille.

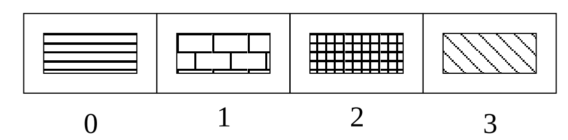
— Chargement dans le cache de blocs constitués d'emplacements contigus de la mémoire lente tire partie de la localité spatiale

— Un même bloc peut se trouver à un instant donné à la fois dans le cache et dans la mémoire lente

Exemple 8. — Caches matériels : blocs de quelques octets

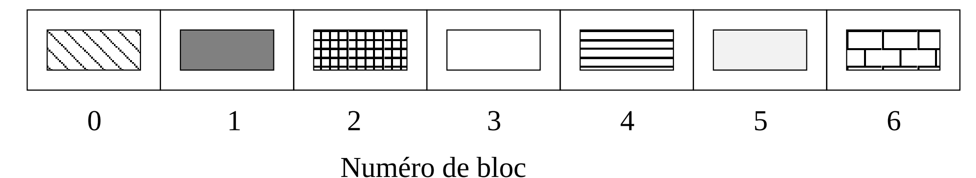
— Caches disques : blocs de quelques secteurs

Mémoire rapide − "cache"



Numéro de case (emplacement dans mémoire cache)

Mémoire lente



Numéro de bloc

— *Fonction de correspondance* permettant de savoir où se trouve un bloc dans le cache et s'il y est chargé

Exemple 9. — Caches matériels : fonction de correspondance simple (accès en un cycle) - voir plus loin
— Caches logiciels : fonctions de correspondance peuvent être plus complexes (structures de données telles que des arbres, des tables de hachage, etc)

Déroulement d'un accès à l'information :
— la donnée demandée *appartient au cache* : on y accède directement dans le cache (en lecture ou écriture)
— la donnée demandée *n'appartient pas au cache* : chargement dans le cache à partir de la mémoire lente, en tenant à jour la structure de donnée décrivant l'état du cache ; puis accès dans le cache

Mise en œuvre : politique de recopie
— *Constat* : si on modifie un élément du cache, il y a une différence entre la version de la donnée correspondante dans le cache et en mémoire lente (version en cache la plus récente)
— On dit dans ce cas que la version en cache est *modifiée*
— *Pourquoi recopier ?*
  — rendre les modifications *durables*
  — anticiper l'éviction de ce bloc du cache
— *Types de recopie*
  — *Recopie simultanée* ("write-through") : on recopie l'information à chaque écriture dans le cache
  — *Recopie différée* ("write-back") : on recopie l'information "plus tard", mais dans tous les cas avant de l'effacer de la mémoire rapide lors d'un remplacement

Mise en œuvre : politique de remplacement
— *Pourquoi remplacer ?* : le cache est plus petit que la mémoire lente, il va se remplir rapidement
— Que faire lors d'un défaut de cache lorsque le cache est plein ? charger la donnée manquante dans un emplacement occuppé auparavant par une autre donnée :

1. *réquisitionner* un bloc du cache : recopie de l'information en mémoire lente si nécessaire plus modification de la structure de données décrivant l'état du cache
2. *charger* l'information manquante plus modification de la structure de données décrivant l'état du cache

— Choix du bloc à réquisitionner (*politique de remplacement*) :
  — Pas de stratégie optimale : sans connaissance de l'avenir, on ne peut pas savoir à coup sûr quelle information sera réutilisée dans le futur
  — LRU (Least Recently Used) : réquisition du bloc accédé le moins récemment
  — Random (choix aléatoire)
  — Limiter la durée de remplacement en évitant de réquisitionner des cases "modifiées" (on évite ainsi la phase de recopie avant le chargement)
— Complexité de mise en œuvre influence le choix d'une stratégie de remplacement

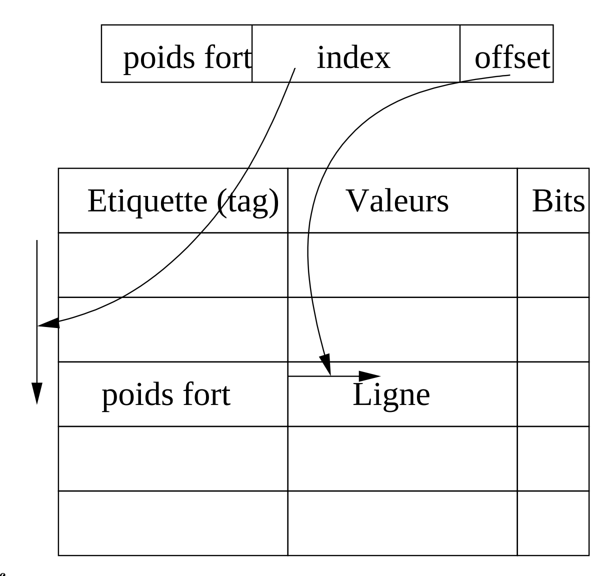
2.4 Caches matériels

Caches matériels : état du cache

— Unité de transfert des informations depuis la mémoire : *bloc* de quelques octets
— Adressage d'un bloc : adresse mémoire
— Contraintes sur le placement d'un bloc dans le cache :
  — Un seul emplacement possible : *correspondance directe* (*direct mapped*)
  — N'importe où dans le cache : caches *totalement associatifs* (*fully associative*)
  — Dans un nombre fixe limité d'emplacements : *associatif par ensemble* (*set associative*)

Correspondance directe

— Fonction de correspondance Emplacement = @ MOD nb_blocs
— Bits
  — validité du bloc (V)
  — modification (M)
— Etiquette (tag) : indication du contenu
— Présence dans le cache : V=1 et tag=@ recherchée



Totalement associatifs

— Bloc à n'importe quel emplacement du cache
— Recherche dans tous les emplacements du cache en parallèle
— Etiquette : contient quasiment toute l'adresse

— En pratique, caches de petite capacité (recherche associative)

Adresse

poids fort	offset

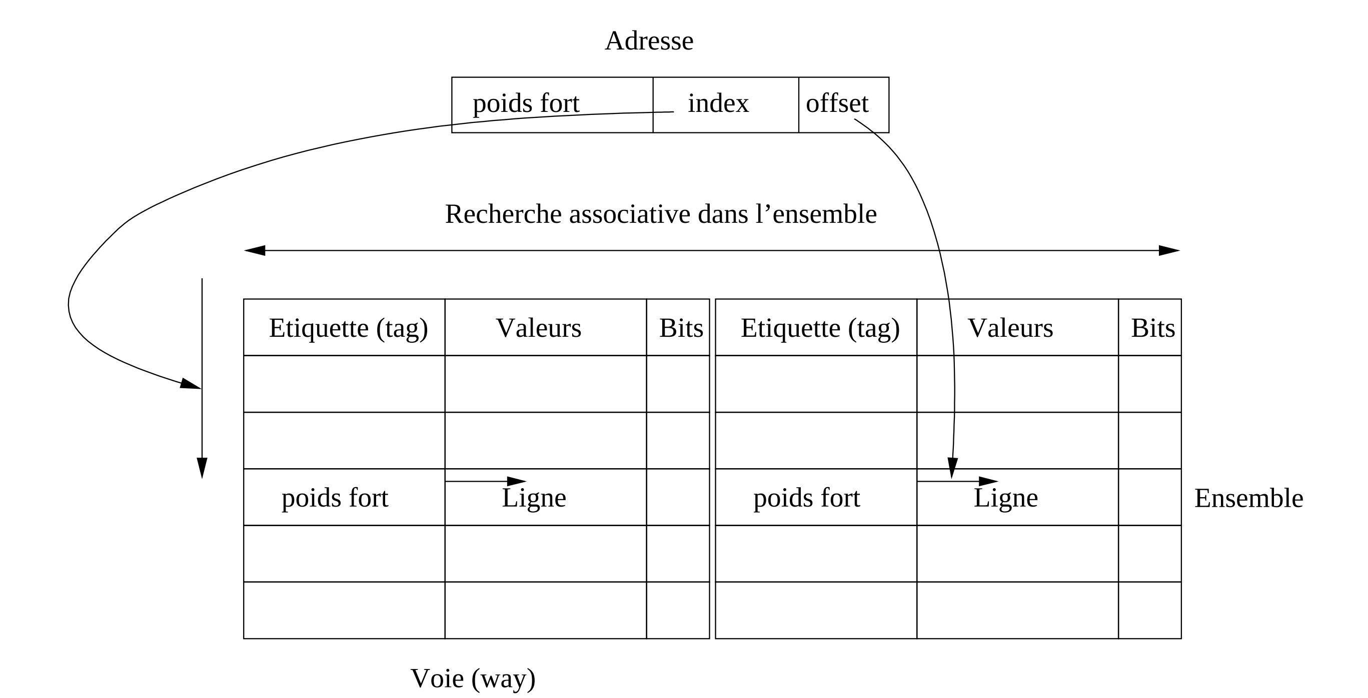
Etiquette (tag)	Valeurs	Bits
poids fort	Ligne	

Recherche totalement associative

Associatifs par ensemble

- *Ensemble* (*set*) : groupe de blocs dans le cache
- *Degré d'associativité* (*associativity degree*) : nombre de blocs par ensemble
- Si n blocs dans un ensemble, on parle de caches *associatifs par ensemble de 2-voies* (*n-way set associative*)
- Fonction de correspondance :
  1. sélection de l'ensemble, soit en général ensemble = @ MOD nb_ensembles
  2. recherche en parallèle du bloc dans l'ensemble
- Très utilisé en pratique

Cache associatif par ensemble de 2 voies



Caches matériels : politique de remplacement

Politiques les plus courantes :

— *LRU* (Least Recently Used) : remplacement du bloc qui a été accédé depuis le plus longtemps
— *FIFO* (First In, First Out)
— *Random*

Caches matériels : politique de recopie

— Types de recopie
  — *Recopie simultanée* ("write through") : on recopie l'information à chaque écriture dans le cache
  — *Recopie différée* ("write back") : on recopie l'information "plus tard", mais dans tous les cas avant de l'effacer de la mémoire rapide lors d'un remplacement
— Types d'allocation
  — *Allocation en écriture* ("write allocate") : une écriture s'effectue dans le cache (chargement puis écriture)
  — *Ecriture sans allocation* ("nowrite allocate") : le bloc est directement modifié dans le niveau inférieur
— *Combinaisons courantes* : *write allocate + write back* ou *nowrite allocate + write through*

Caches matériels : répartition des caches

— *Hiérarchies de caches*
— Caches d'instructions et caches de données *séparés* ou *unifiés* (en règle générale, caches L1 séparés)
— Problèmes de *cohérence* en caches en multiprocesseurs

2.5 Caches logiciels

- Omniprésents
- Caches du *contrôleur disque* (éviter les accès disque)
- Caches du *SGF local* (éviter les accès disque, notamment sur les répertoires)
- Caches d'un *SGF réseau* (ex : NFS, Network File System) : éviter la latence réseau et les accès disque
- Caches *Web*
- Mémoire virtuelle et pagination à la demande, etc.

Deuxième partie

Adressage virtuel et pagination

1 Introduction

Objectifs

- Premier semestre : éléments chargés de gérer le processeur et les entrées/sorties
- Deuxième semestre : éléments chargés de gérer la mémoire
- Accent mis sur les points suivants :
  - mémoires paginées
  - liens existants avec gestion processeur et E/S

Concepts et notations

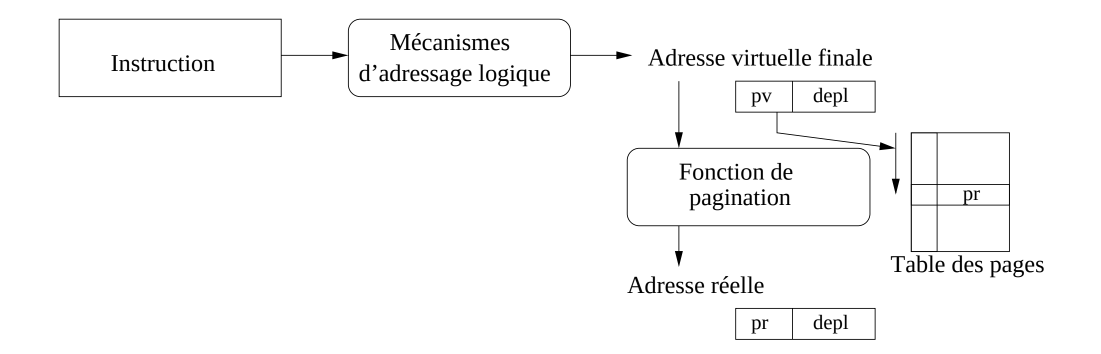
- *Espace virtuel* (ou espace logique) : ensemble des informations accessibles par un processeur virtuel exécutant un processus
- *Adresse virtuelle* : adresse dans l'espace virtuel d'un processus
  Exemple 10 (Adresse virtuelle). Un processeur 32 bits peut accéder à 4 Go d'espace mémoire, indépendamment de la capacité de mémoire physique disponible. Les adresses virtuelles vont de 0 à 2^{32}-1
- *Adresse physique* (ou réelle) : adresse dans la mémoire vive de la machine
  Exemple 11 (Adresse physique). Les adresses physiques sur un processeur doté de 128 Mo de mémoire vont de 0 à 2^{27}-1
- Objets manipulés par les programmes : objets *logiques* (variables, procédures, "objets")
- Mise en œuvre finale de ces objets logiques en utilisant des objets *physiques* (emplacement en mémoire physique)
- Passage des objets logiques aux objets physiques en deux phases :
  - implantation des objets logiques dans l'espace virtuel (liaison *logique*, i.e. attribution d'adresses virtuelles) : voir partie 4
  - mise en œuvre de cet espace virtuel sur des supports physiques (adresses physiques, disque). C'est l'objectif de ce chapitre

- gérer l'*allocation* des emplacements logiques sur les supports physiques (disque, mémoire centrale)
- gérer les *transferts* entre les différents supports
- réaliser la *correspondance* entre adresses virtuelles et adresses physiques

2 Mécanisme de pagination

Introduction au mécanisme de pagination

- Découpage de l'espace virtuel en morceaux de taille fixée, nommés *pages*
- Adresse virtuelle = couple (n^o de page virtuelle, déplacement dans la page virtuelle).
- Le mécanisme d'adressage remplace, à chaque accès, le numéro de page virtuelle par un numéro de page physique



Intérêts du mécanisme de pagination

- — Permet la réimplantation dynamique des programmes
- — Utilisation de programmes dont la taille cumulée dépasse celle de la RAM
- — Permet le va-et-vient RAM-disque automatique par page

	0
	1
	2
	3
	4

Table des pages



	1000
	1001
	1002
	1003
	1004

Table des pages

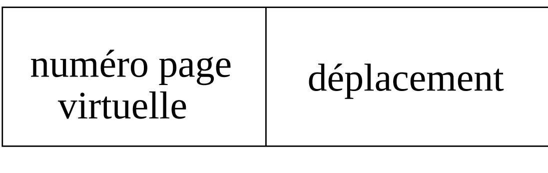
Présentation des notions :

1. *Cas simples*
   - — espace virtuel linéaire
   - — fonction de pagination simple (simple table de correspondance)
2. *Extensions*
   - — fonctions de pagination plus élaborées
   - — accélération des accès mémoire, etc.
3. Autres mécanismes permettant la réimplantation dynamique

2.1 Adresse virtuelle et adresse physique

Adresse virtuelle

- — Découpage de l’espace virtuel en *pages virtuelles* (pages *logiques*) de taille fixe
- — Une *adresse virtuelle* peut être interprétée comme un couple *n^o de page virtuelle, déplacement dans la page*
- — Si les pages font 2^m octets, et s’il y a 2^v pages virtuelles, une adresse virtuelle fait $m + v$ bits, organisés de la manière suivante :



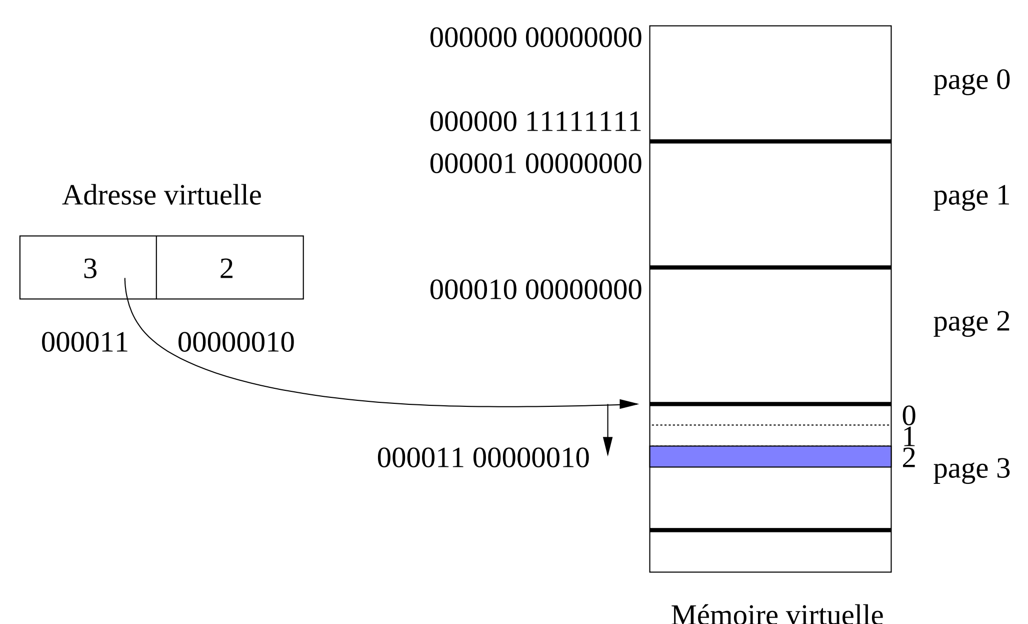
— On notera une adresse virtuelle (pv, d) uniquement pour plus de lisibilité, en réalité, c'est une simple suite de bits

Remarque 12. — *L'ensemble des pages virtuelles constitue un espace d'adressage unique : le dernier mot de la page i est suivi du premier mot de la page* $i + 1$

— *Par exemple, si une page fait* 2^8 *octets et s'il y a* 2^6 *pages, l'adresse a=(2,255) est suivie de l'emplacement d'adresse a+1 = (3,0). a= 000010 11111111, a+1 = 000011 00000000*

Adresse virtuelle : exemple

Système avec pages de 2^8 (256) octets, 2^6 pages

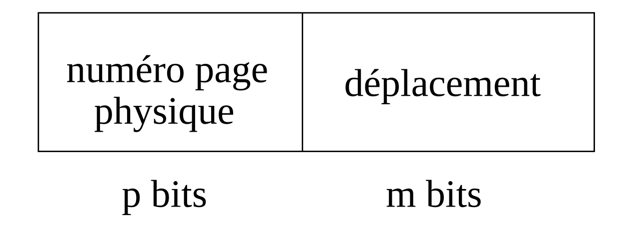


Adresse physique

— Découpage de la mémoire physique en *pages physiques* (pages *réelles*) de taille fixe

— Une *adresse physique* (adresse *réelle*) peut être interprétée comme un couple *n^o de page physique, déplacement dans la page*

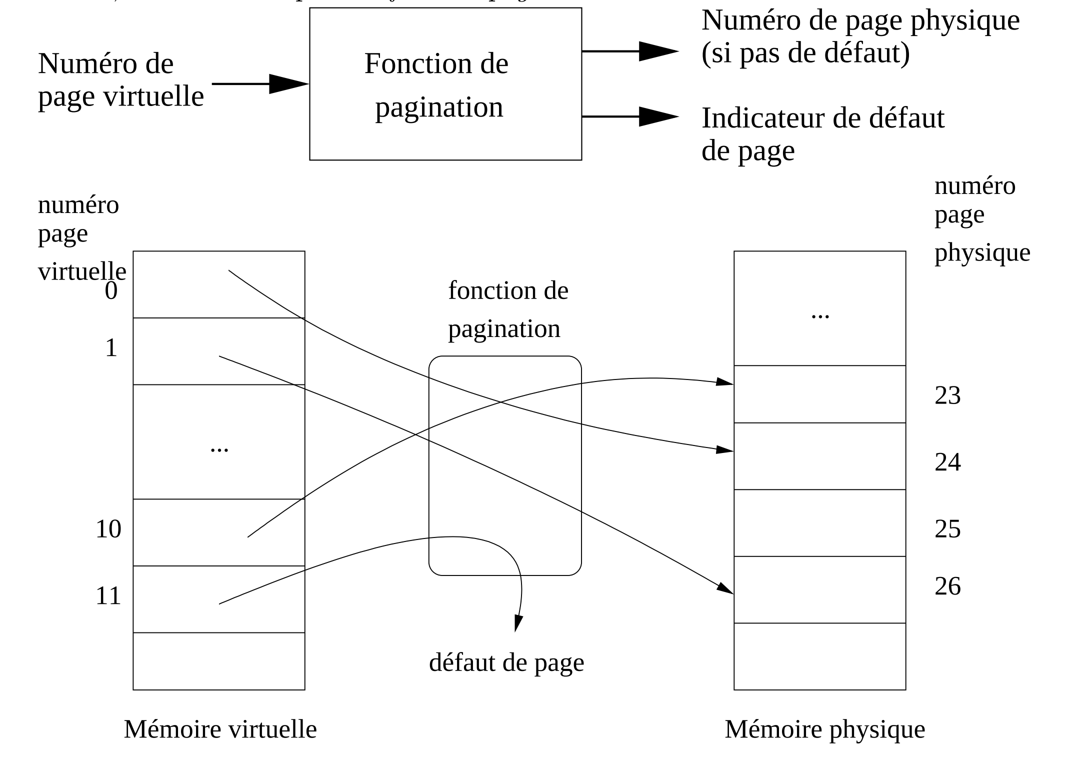
— Si les pages font 2^m octets, et s'il y a 2^p pages physiques, une adresse physique fait $m + p$ bits, organisés de la manière suivante :



Remarque 13. *En général :*
- *On a $v \neq p$, et $v >> p$*
- *Taille des adresses virtuelles >> taille des adresses réelles*
- *Espace virtuel adressable >> espace physique disponible*

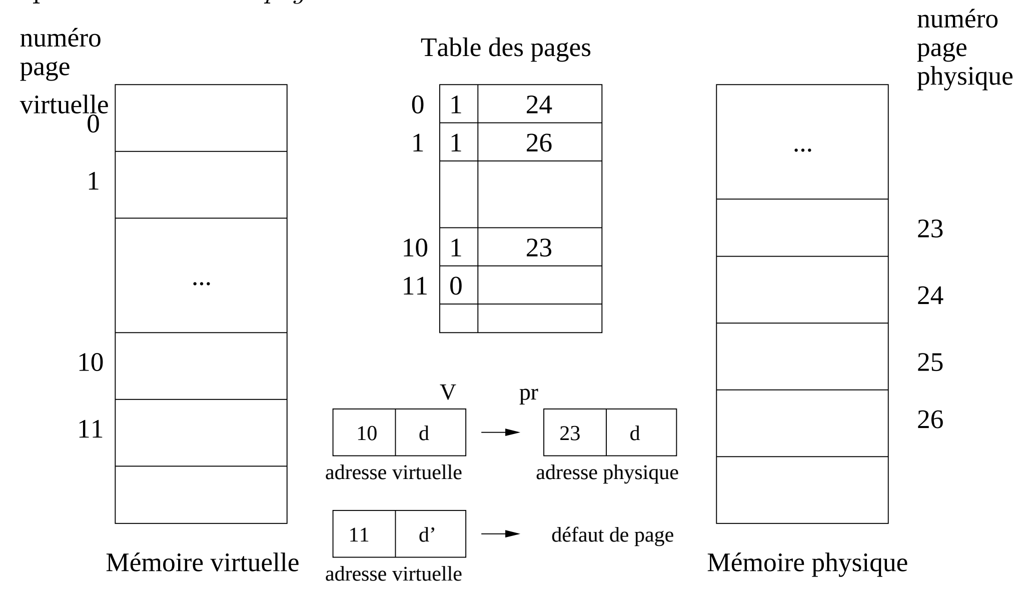
2.2 Fonction de pagination

- Mise en œuvre par *matériel* : Unité de Gestion mémoire (UGM), Memory Management Unit (MMU)
- Transforme à chaque accès mémoire un numéro de page virtuelle en numéro de page physique
- Fonction *non totale* : il peut y avoir des adresses virtuelles non traduites par la fonction de pagination. Il s'agit alors d'une exception signalée au système : *défaut de page*
- Deux issues lors d'un appel à la fonction de pagination :
  - la traduction est possible : retourne l'adresse physique résultat
  - sinon, *déroutement pour défaut de page*



Fonction de pagination : mise en œuvre par table des pages linéaire

- *Tableau* avec une entrée par page virtuelle (n^o page = indice du tableau)
- Bit *de présence* (bit V) indiquant si une page physique est associée à la page virtuelle (sinon, déroutement pour défaut de page)
- Table stockée en mémoire
- On parle de *table des pages linéaire*



- *Ralentissement de l'exécution* par rapport à un système sans adressage virtuel. Pour chaque accès à une adresse virtuelle, au moins *deux* accès à la mémoire physique :
  1. Un accès à la table des pages, située en mémoire, pour récupérer le numéro de page physique
  2. Un accès à l'emplacement contenant l'information proprement dite
- Table des pages en mémoire $\Longrightarrow$ problème d'*espace mémoire* occupé et de *temps d'accès*. Mécanismes de traitement de ces problèmes vus plus loin.
- Format typique d'une entrée de la table des pages (DPV, *descripteur de page virtuelle*)
  - *présent* (V) : indique si une page physique est associée à la page virtuelle
  - *droit* : bits spécifiant les droits d'accès à la page (lecture, écriture, exécution)
  - *pphys* : numéro de la page physique associée (si présent=1)
  - et d'autres informations vues plus loin ...

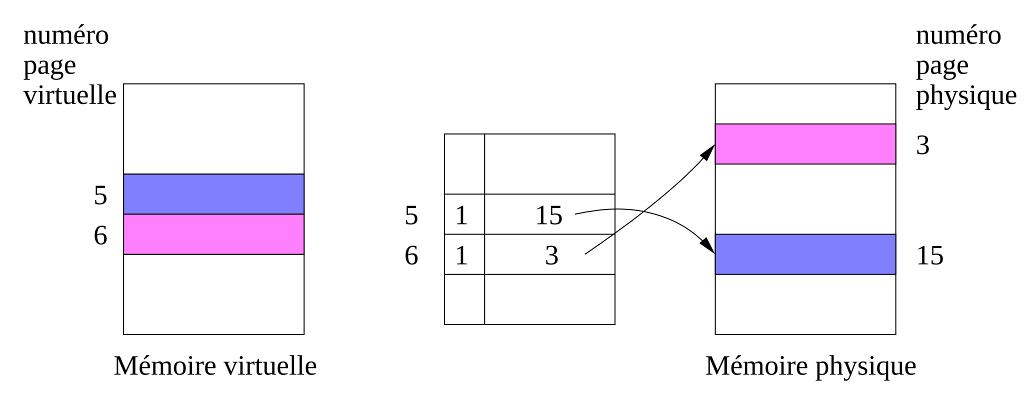
"Code" de la fonction de pagination (réalisée par *matériel*) :

```
type adVirt = (v bits pv, m bits d)
     adPhys = (p bits pp, m bits d)
     DPV = ( bit present, bits droit,p bits pphys)
     typeAcces = {lire, écrire, ...}
var [0 :2^v-1]DPV tpages; {table des pages, adresse dans registre MMU}
function pagination (adVirt adv, typeAcces acces) resultat adPhys
begin
     if acces incompatible avec tpages[adv.pv].droit
        then déroutement pour violation de protection mémoire
        else if tpages[adv.pv].present = 0
           then déroutement pour défaut de page
           else return (tpages[adv.pv].pphys, adv.d)
```

```
        end if
    end if
end
```

Contiguité des pages en mémoire

Remarque 14. *Des pages contiguës dans l'espace virtuel ne le sont pas obligatoirement dans l'espace physique.*



- Dans le cas d'un adressage indirect avec relais en mémoire, le relais contient une *adresse virtuelle*. Pour accéder à l'opérande final on passe donc *deux fois* par le mécanisme de pagination :
  - une première fois lors de l'accès au relais
  - une seconde fois lors de l'accès à l'opérande
- Transformation d'adresse à l'exécution lors de chaque accès $\Longrightarrow$ liaison n'est terminée qu'au dernier moment $\Longrightarrow$ permet la réimplantation dynamique (simple mise à jour des tables de traduction)
- Un programme peut être "partiellement présent" en mémoire physique : certaines pages sont effectivement présentes en mémoire centrale, d'autres ne le sont pas
- Les espaces virtuels et physiques n'ont pas forcément la même taille. Si on a 2^v pages virtuelles et 2^p pages physiques :
  - Si $v > p$, l'espace virtuel ne tient pas entièrement en mémoire physique. On verra que l'on peut exécuter quand même de tels programmes
  - Si $v < p$, on peut mettre plus d'un espace virtuel en mémoire physique (un seul accessible à la fois, mais plusieurs peuvent être résidents)
- Protection à l'exécution, espaces mémoire séparés

3 Pagination à la demande

3.1 Principe

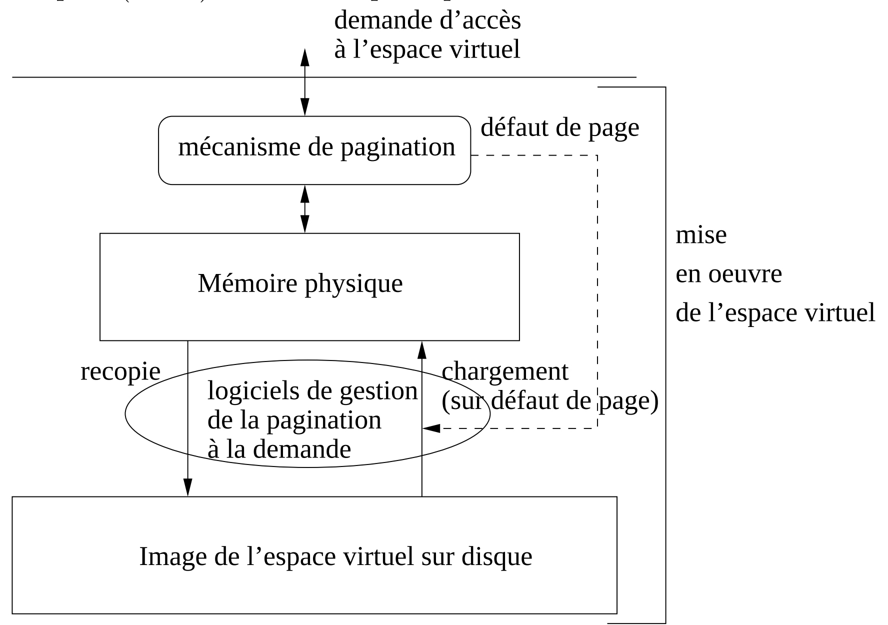
- Fonction de pagination $\Longrightarrow$ non contiguité
  - $\Longrightarrow$ Allocation mémoire simplifiée. Il suffit de trouver n pages, quelles que soient leurs adresses
  - $\Longrightarrow$ Mise en œuvre d'un mécanisme de *va-et-vient global* aisée (changement contenu tables de traduction)

— Constats :
  — Localité des applications $\Longrightarrow$ à un instant donné, une application n'a besoin que d'un sous-ensemble de ses informations
  — Le mécanisme de pagination permet de n'avoir en mémoire qu'un sous-ensemble des pages virtuelles

$\Longrightarrow$ Chargement d'une page en mémoire physique que si elle est référencée : *pagination à la demande*

$\Longrightarrow$ Au lieu d'un va-et-vient global sur tout le programme, on peut effectuer un *va-et-vient au niveau de la page* en fonction des besoins

Principe du cache :
— mémoire lente = espace virtuel, dont on a l'image sur disque
— mémoire rapide (cache) = mémoire principale de la machine



— Initialement, aucune page en mémoire physique (espace virtuel sur disque, cache vide)
— Accès à une page non présente $\Longrightarrow$ *déroutement pour défaut de page*. La routine exécutée doit rendre possible l'exécution de l'instruction fautive :
  1. trouver une page physique disponible
  2. la remplir avec l'image disque de la page virtuelle
  3. modifier la table des pages pour noter la présence de la page virtuelle en mémoire physique
  4. ré-exécuter l'instruction fautive

Éléments de mise en œuvre

Identiques aux problèmes à résoudre pour les mémoires cache :
— *Représentation de l'état du cache* : supportée par structure de donnée de la fonction de pagination (au plus simple, table des pages linéaire)
— *Représentation de l'état de la mémoire lente* (disque)
— *Politique de recopie* : quelle stratégie adopter pour la recopie des pages modifiées ?
— *Politique de remplacement* : quelle page supprimer du cache quand il est plein ?

3.2 Éléments de mise en œuvre

Représentation de l'image sur disque

- Principe : associer une image sur disque à chaque page virtuelle
- Utilité :
  - Savoir à quel emplacement lire une page virtuelle lors de son chargement (défaut de page)
  - Savoir à quel emplacement recopier une page virtuelle modifiée (recopie)
- Emplacements possibles de stockage de l'adresse disque :
  - Dans le descripteur de page virtuelle (table des pages)
  - Dans une table séparée, ayant une entrée par page virtuelle

Zones du disque dédiées au stockage de l'image disque des pages virtuelles :

- Zones non modifiables : *fichier exécutable*
- Zones modifiables : *zone d'échanges* (ou *swap*) (partition, fichier)

Instants de mise en correspondance d'une page virtuelle et de son image disque :

- Correspondance *statique* (au chargement) : au "chargement" d'un programme, on alloue l'image de toutes ses pages virtuelles modifiables dans la zone d'échanges
- Correspondance *dynamique* (à l'exécution) : on établit la correspondance *au plus tard*, lors de la recopie d'une page virtuelle sur disque. Intérêts de la correspondance dynamique :
  - On n'alloue sur disque que ce qui est strictement nécessaire (utilité pour les pages de pile)
  - On peut optimiser les déplacements du bras en plaçant intelligemment les données sur disque

Instants de libération de l'image disque :

- Fin de programme

Stratégie de recopie

- *Recopie immédiate* (write-through) : beaucoup trop coûteux et inutile (information en mémoire n'est pas permanente)
- D'où *Recopie différée* (write-back). Quand recopier ?
  - *Lors d'un remplacement de page*, quand une page est supprimée de la mémoire et qu'elle est modifiée
    - $\Longrightarrow$ Deux E/S disque pour le traitement du défaut de page (recopie de la page requisitionnée si modifiée + lecture de la page manquante depuis le disque)
  - *De manière décorrélée avec le remplacement de page*, par un processus indépendant
    - $\Longrightarrow$ On peut réquisitionner uniquement les pages non modifiées
    - $\Longrightarrow$ On risque d'effectuer des recopies inutiles

Remplacement de page

- Forte probabilité qu'au bout d'un certain temps, il n'y ait *plus de page physique disponible* dans le système
- Il faut alors lors d'un défaut de page *réquisitionner* une page pour l'attribuer au processus en défaut (*remplacement de page*)
- Chronologie (défaut sur page *pv1*) :
  1. Sélection de la victime *pr* (*réquisition*), la page *pr* va être *vidée*. La mémoire étant pleine, *pr* supporte déjà une page virtuelle *pv2*.

2. On note dans son descripteur que *pv2* n'est plus présente
3. Recopie de *pv2* si nécessaire (dépend de la stratégie de recopie)
4. Reste du traitement de défaut de page *pr* est maintenant disponible

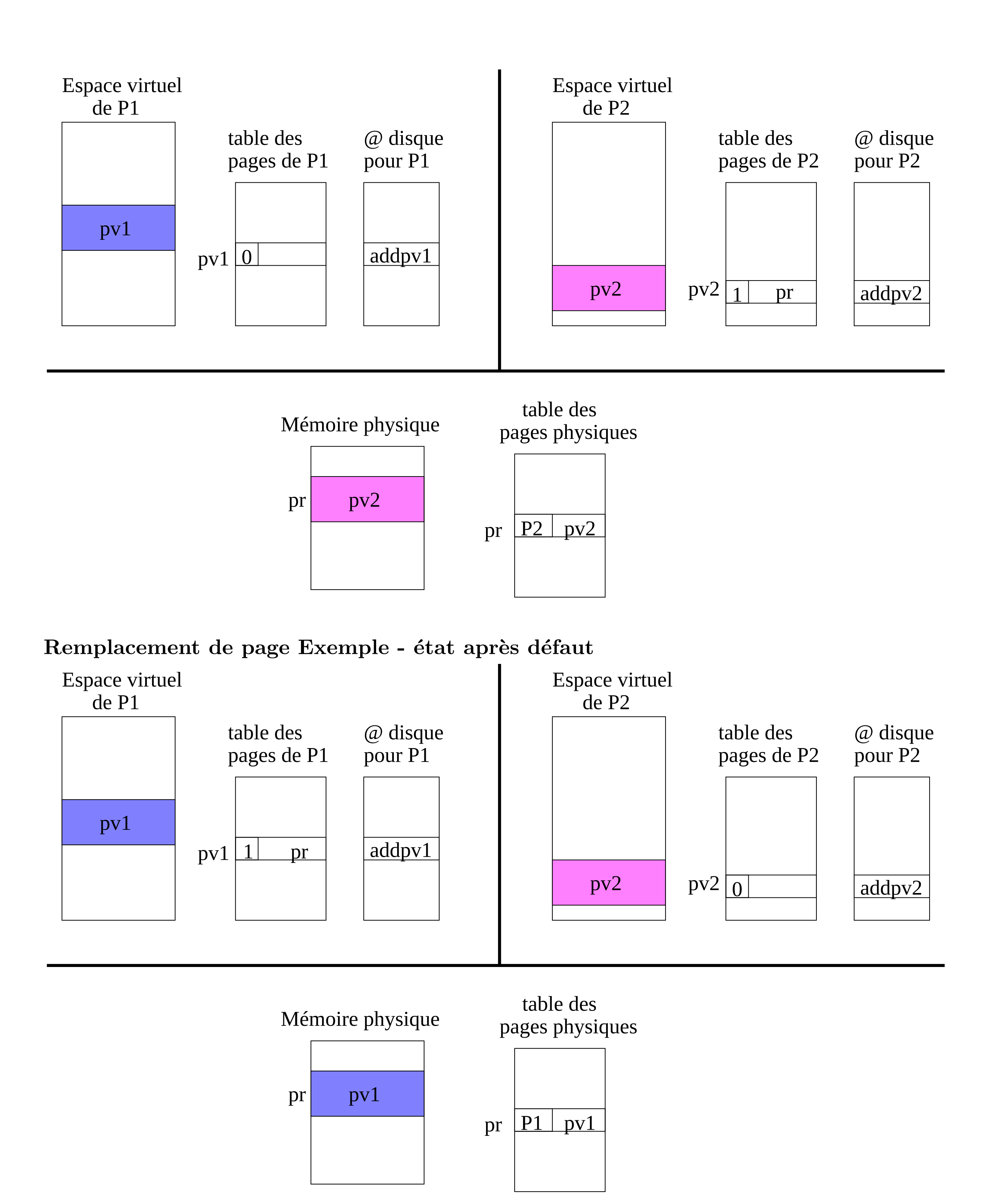
Remplacement de page Structures de données

— Nécessaire d'avoir un *état d'allocation des pages physiques* : occupées ou libres, et si occupées, où se trouvent les informations relatives à la page virtuelle supportée (descripteur, adresse disque)
— Ces informations sont dans un *descripteur de page physique*, contenant :
  — Un *lien inverse* vers le descripteur de page virtuelle supportée : pointeur, ou couple (processus propriétaire, numéro de page virtuelle)
  — Un bit de *modification* (M) indiquant si la page a été modifiée depuis son chargement en mémoire
  — un bit d'*utilisation* (U) indiquant si la page a été référencée

Remplacement de page Exemple

1. L'ARP choisit la page physique *pr* pour faire le remplacement
2. l'entrée *pr* de la table des pages physiques fournit *(P2, pv2)*
3. on met l'entrée *pv2* de la table des pages de *P2* à *(0,-)*
4. l'entrée *pv1* de la tables des adresses disque de *P1* fournit *addpv1*
5. on fait une lecture disque depuis *addpv1* vers la page physique *pr*
6. on met l'entrée *pv1* de la table des pages de *P1* à *(1, pr)*
7. on met l'entrée *pr* de la table des pages réelles à *(P1, pv1)*

Remplacement de page Exemple - état avant défaut



Remplacement de page Exemple - état après défaut

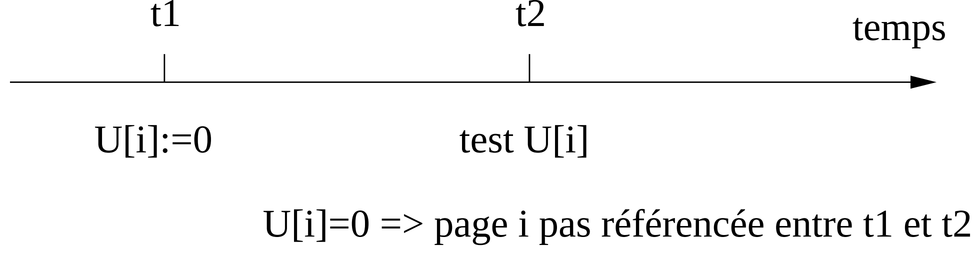
Remplacement de page Algorithme de remplacement de page

Definition 15 (Algorithme de remplacement de page)**.** On nomme *algorithme de remplacement de page* (ARP) l'algorithme de sélection d'une page à réquisitionner lors d'un défaut de page lorsqu'il n'existe plus aucune page disponible

— Mesure de l'efficacité de l'ARP = taux de défauts de page (nombre de défauts / nombre de références)
— ARP optimal (algorithme de Belady) n'est pas réalisable en pratique (nécessite de connaître le futur)
  $\Longrightarrow$ Algorithmes utilisés utilisent les références passées et les propriétés de localité des programmes

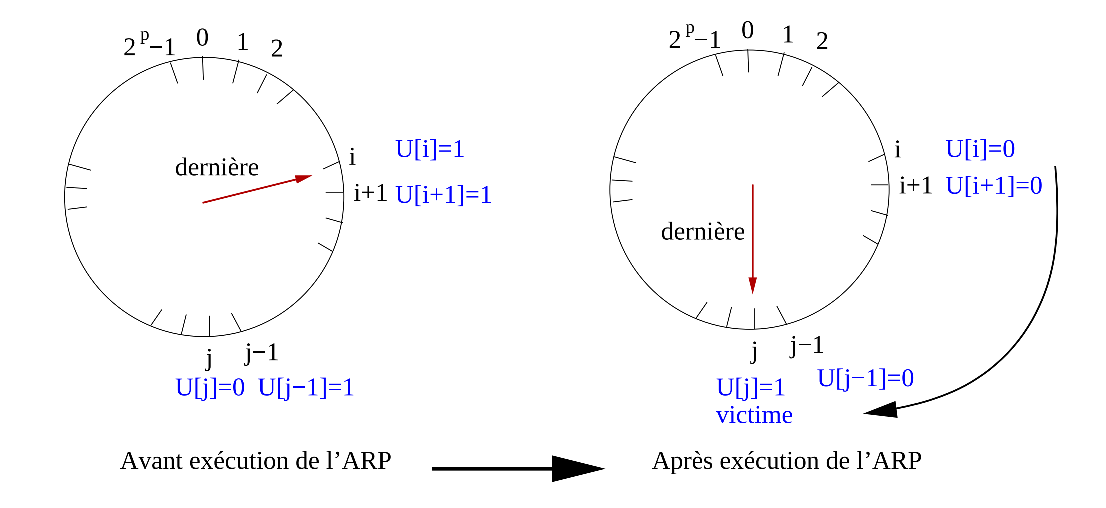
Remplacement de page Algorithmes de remplacement de page classiques

- *FIFO* (First In First Out) : sélection de la page la plus anciennement chargée
  - ne respecte pas le principe de localité : une page souvent référencée sur une longue période finira par être la plus ancienne, et donc sera vidée
  - facile à mettre en œuvre
- *LRU* (Least Recently Used) : la page victime est celle dont la dernière référence est la plus ancienne
  - utilise la propriété de localité : les pages utilisées récemment ne sont pas vidées
  - difficile à mettre en œuvre : il faudrait maintenir une liste des pages virtuelles présentes triée par date de dernière référence (liste mise à jour *à chaque référence*)
- Algorithme de la *seconde chance* (horloge) : approximation de l'algorithme LRU, qui choisit comme victime une page non référencée "récemment" (pas nécessairement la plus anciennement référencée)
  - Utilisation du bit U mis à 1 à chaque référence à la page. U=1 signifie que la page a été référencée "récemment". Si à t1 on remet à 0 le bit U de la page i, la valeur de ce bit à t2>t1 permet de décider si la page a été utilisée ou non entre t1 et t2



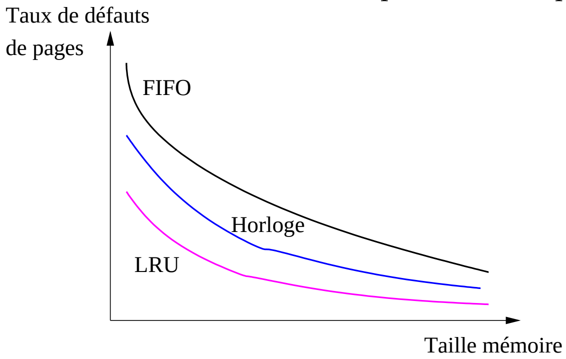
  - on ordonne les pages physiques circulairement par rapport à leur numéro, en conservant le numéro (noté *dernière) de la dernière page vidée*
  - *lors d'une demande de remplacement, parcours de la liste des pages physiques à partir de* dernière*, jusqu'à avoir trouvé une page dont le bit U est à 0. Pour toutes les pages entre dernière et la victime choisie, on remet le bit U à 0, ce qui assure que l'algorithme fournira bien une réponse dans tous les cas*

Exemple d'exécution de l'algorithme de l'horloge ($\forall k \in [i, j-1] U[k] = 1$))



Remplacement de page Performances des algorithmes

Performances obtenues sur des chaînes de références provenant d'applications réelles



"Chargement" d'un programme

- Contenu d'un fichier exécutable :
  - Description (adresse en mémoire et contenu) des zones contenant du code et des données initialisées
  - Description (adresse en mémoire et taille) des zones contenant des données non initialisées
- Objectif du chargement : Initialiser la mémoire pour que le programme puisse commencer à s'exécuter
- *Sans pagination* (travail directement en adresse physique) : consiste à *implanter* le programme en mémoire à partir du fichier exécutable
  - réservation de mémoire pour les différentes zones
  - remplissage de ces zones à partir du disque
  - initialisation des registres du processeur (SP,PC)
- *Avec pagination à la demande* : pas de chargement en mémoire avant le début de l'exécution, mais à la place *initialisation des tables des pages*. Le chargement se fera lors des défauts de page
  - bit *présent* = 0 (toutes les pages sont absentes)
  - initialisation des adresses disque pour référencer la zone d'échanges
  - initialisation des registres du processeur (SP,PC)

Trois types de zones vis à vis du chargement :

— *code* : son image reste dans le fichier exécutable (non modifiable)

— *données initialisées* : leur état initial doit être obtenu à partir du fichier exécutable, mais leur image sera ensuite sur la zone d'échange (une copie *par processus*)

— *données non initialisées* et *pile* : pas d'état initial fixé, leur image sera tout le temps dans la zone d'échange.

"Chargement" d'un programme Solution simple

— Principe
  — allocation *statique* (au chargement) de l'image disque
  — initialisation de la partie de la zone d'échanges correspondant aux données initialisées

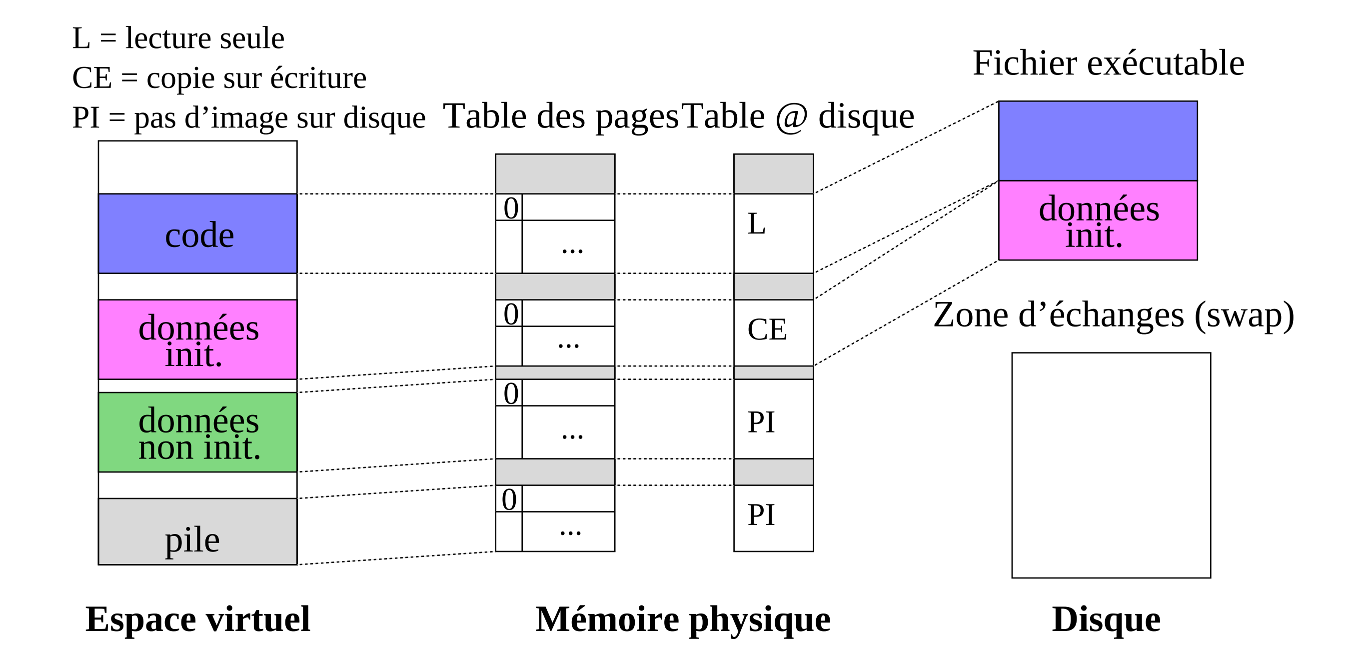
— Chronologie

1. réservation d'espace virtuel (table des pages) pour les différentes zones (code, data, bss, stack)
2. réservation disque dans la zone d'échanges pour data, bss, stack
3. recopie de l'état initial des données initialisées (fichier exécutable) vers la partie de la zone d'échange correspondante
4. initialisation de la table des pages (V=0) et la table des adresses disques

"Chargement" d'un programme Améliorations

— Allocation *dynamique* d'espace disque pour les données non initialisées et la pile. Allocation au plus tard (lors du remplacement de page)

— Allocation *paresseuse* d'espace disque pour les données initialisées. Tant qu'une page de cette zone n'est pas modifiée en mémoire, on continue à utiliser l'image disque du fichier exécutable. Allocation d'espace disque au plus tard (lors de la recopie)

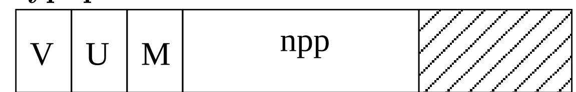


Zones licites vs illicites

Definition 16 (Zone illicite)**.** Zone d'adresses virtuelles dont l'accès entraîne une erreur à l'exécution. Zone licite = zone pour laquelle les tables des pages sont allouées.

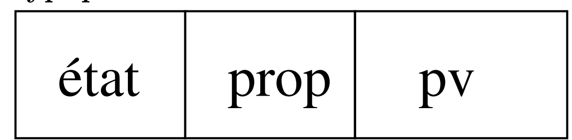
Repérage des zones illicites :
- Bits non gérés par le matériel dans les DPV
- Test lors de défauts de page

Récapitulatif : contenu typique d'un DPV



- V : test de résidence
- U : pour remplacement de page
- M : pour recopie
- npp : emplacement en mémoire
- hachuré : ignoré par MMU (test si licite, copy-on-write, etc.)

Récapitulatif : contenu typique d'un DPR



- état : état d'allocation (libre, occuppé, verrouillé)
- prop : espace d'adressage propriétaire de la page
- pv : numéro de la page dans l'espace d'adressage
- (prop,pv) : forment un lien inverse pour accès DPV

4 Amélioration des performances

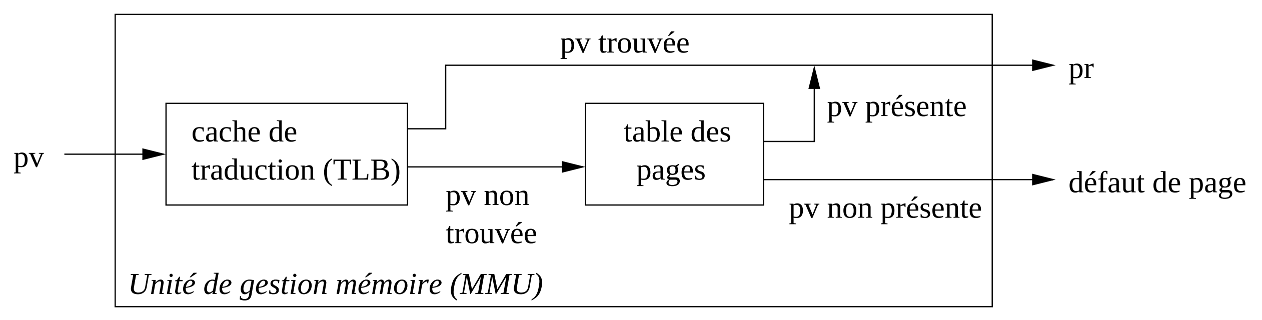
4.1 Caches de traduction

- Introduction du mécanisme de pagination $\Longrightarrow$ *ralentissement important des accès mémoire*
  - à chaque accès, accès mémoire proprement dit + lecture du descripteur de la page virtuelle (plusieurs cycles mémoire peuvent être utiles selon sa taille)
- Amélioration possible basée sur la propriété de localité et le principe du *cache* appliqué à la table des page (pendant une période assez longue, le programme va référencer un petit sous-ensemble de ses pages)

Definition 17 (Cache de traduction d'adresses)**.** Un *cache de traduction d'adresses* ou TLB (Translation Lookaside Buffer) est une mémoire cache matérielle contenant les correspondances page virtuelle / page physique les plus utilisées

Déroulement d'un accès à une page virtuelle *pv* :

1. Recherche dans le TLB. Si trouvé, on obtient directement le numéro de la page physique associée *pp*
2. Sinon, recherche dans la table des pages pour trouver la page physique associée *pp* et stockage du couple (pv,pp) dans le TLB (à la place d'un autre couple si le TLB est plein)



Exemple 18. Soit un système avec :
- un temps d'accès à la mémoire, hors pagination, de 100 ns
- un temps d'accès au TLB de 5 ns
- un descripteur de page virtuelle lu en un cycle mémoire
- un taux de succès du TLB de 90%

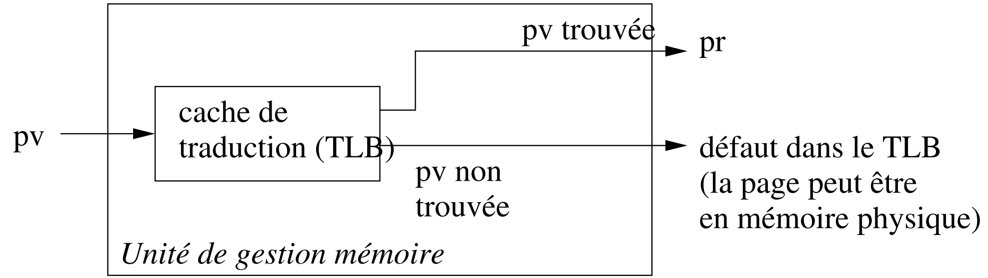
Le temps d'accès mémoire moyen est de 5 + 0.1*100 + 100 = 115ns, contre 200ns sans TLB

Caches de traduction et changements de contexte

- En général, les processus possèdent un espace d'adressage virtuel *privé*
  - $\Longrightarrow$ deux processus différents peuvent utiliser la même adresse virtuelle avec un contenu différent
  - $\Longrightarrow$ mise à jour de la table de traduction courante lors d'un changement de contexte
  - $\Longrightarrow$ contenu du TLB incorrect après un changement de contexte
- Solutions
  - Vidage du TLB lors des changements de contexte
  - Ajout d'un champ ASID (Address Space Identifier) dans le TLB pour éviter le vidage

Architectures avec TLB uniquement

- Le matériel de pagination offre uniquement un cache de traduction, les structures de données pour la fonction de pagination étant alors *entièrement gérées par logiciel*



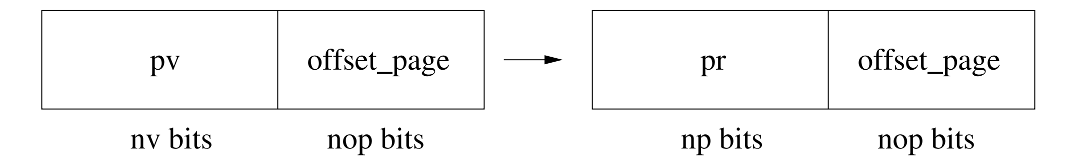
Architectures avec TLB uniquement : MIPS R2000

- Adresses virtuelles et physiques de 32 bits, pages de 512 octets (4 Ko)
- Le CPU contient un TLB de 64 entrées
- Il n'y a pas de table de pages gérée par le matériel
- Table des pages gérée par logiciel. Sur "défaut de TLB", déroutement vers le système d'exploitation qui parcourt la table des pages pour savoir si c'est réellement un défaut de page, puis met à jour le TLB

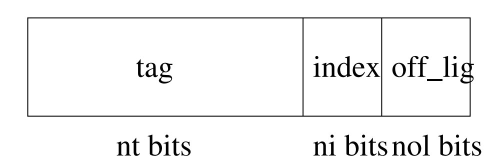
4.2 Mémoire virtuelle et cache

Index et Tags : adresses virtuelles ou réelles ?

Découpage adresse pour pagination



Découpage adresse pour accès cache



- Index :
  - Si $ni + nol < nop$, index cache identique avant et après traduction d'adresse

  $\Longrightarrow$ Indexation en virtuel : *parallélisme* indexation cache/accès TLB
- Tags :
  - En virtuel : on peut accéder au cache sans attendre la translation d'adresse
  - Problèmes : synonymes ou alias (pages logiques projetées sur la même page physique), plusieurs copies de la même donnée dans le cache, problème de cohérence
- En général

— Cache L1 instructions : tags et index en virtuel, ou index en virtuel et tag en réel
— Cache L1 data : index en virtuel, tag en réel
— Cache L2 : tout en réel

4.3 Écroulement du système

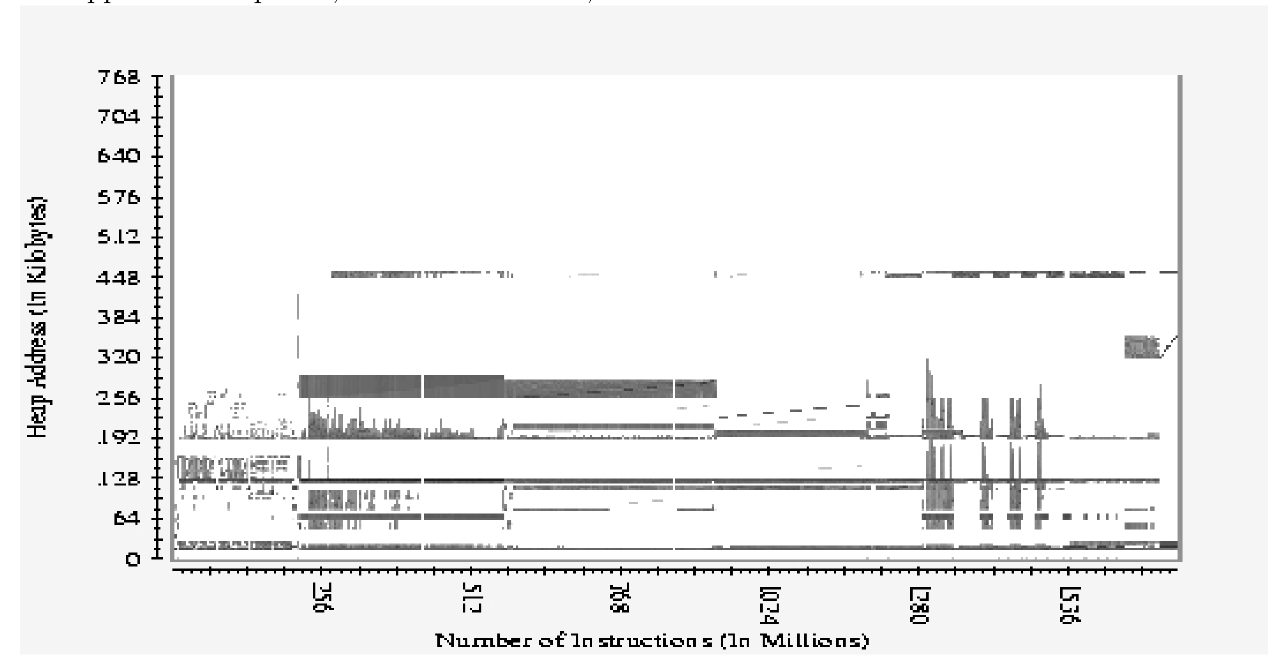
Phénomène d'écroulement

— Premiers systèmes multiprogrammés : diminution brutale des performances quand le nombre d'usagers dépasse un certain seuil : phénomène dit d'*écroulement* (*"trashing"*)
— Le système passe tout son temps à traiter des défauts de page plutôt que d'exécuter les programmes utilisateur

Comportement des programmes

Caractéristiques communes indépendantes des programmes :
— *Non-uniformité* des références aux pages : la fréquence de référence aux pages varie d'une page à l'autre. Un petite partie des pages du programme totalise la plus grande partie des références (ordre de grandeur : 75% des références concernent moins de 20% des pages).
— *Localité temporelle* : pendant une période d'exécution, un processus utilise un sous ensemble réduit de ses pages. Ce sous-ensemble est stable sur la période considérée
$\Longrightarrow$ Phases de *stabilité* relativement longues, utilisant un sous ensemble réduit de pages, séparées par des phases de *transition*, pendant lesquelles le sous-ensemble des pages utilisées change brusquement

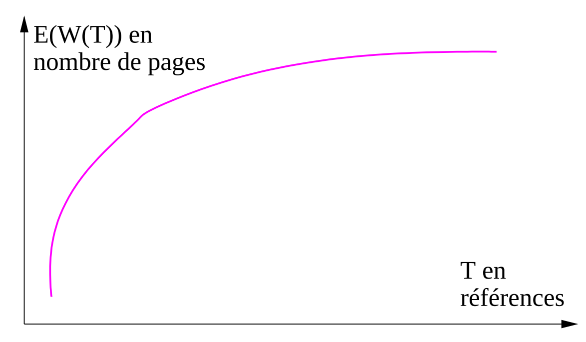
Application Espresso, thèse S. Johnstone, 1997



Comportement des programmes Notion d'ensemble de travail

Definition 19 (Ensemble de travail)**.** Un *ensemble de travail* (*working set*) à un instant t est l'ensemble des pages différentes référencées entre $t-T$ et t. T représente la *largeur de la fenêtre* de

calcul de l'ensemble de travail

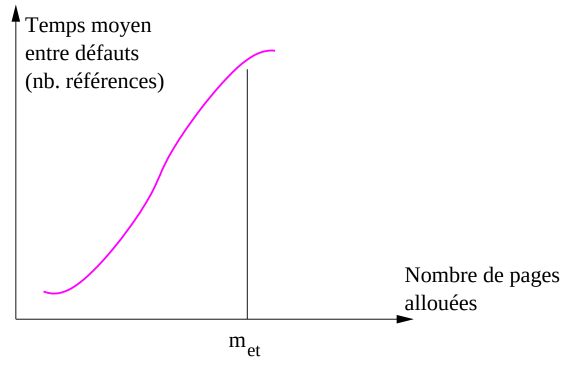


- Quand on augmente la taille de la fenêtre de calcul (T), le nombre de pages différentes référencées croit rapidement puis tend à se stabiliser

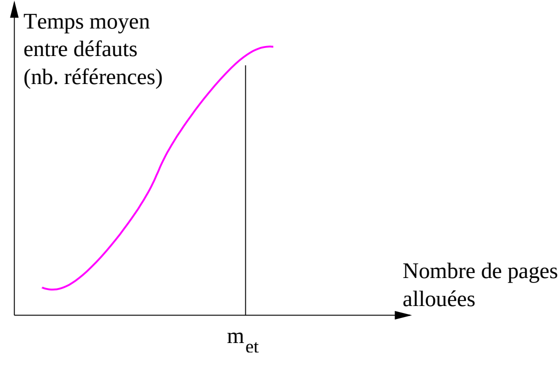
Remarque 20. *Si T est bien choisi (assez grand pour correspondre à la partie asymptotique de la courbe), W(t,T) évolue en général lentement et est une bonne approximation des pages qui seront utilisées dans un futur proche*

Origine du phénomène d'écroulement

Mesure du temps moyen entre défauts en fonction de l'espace mémoire disponible :



- Augmentation rapide jusqu'à un palier
- Palier : espace mémoire suffisant pour loger l'ensemble de travail
- Augmentation de l'espace mémoire au delà de ce seuil est quasiment inutile



- Augmentation du nombre de processus $\Longrightarrow$ diminution de l'espace mémoire disponible par processus
- Si mémoire disponible pour un processus passe en dessous de son m_{et}, alors l'intervalle entre deux défauts chute brusquement

- Conséquences :
  - contrôleur disque saturé, ce qui ralentit d'autant le traitement des défauts de page
  - pendant les E/S on exécute les autres processus, mais eux même déclenchent des défauts de page, etc.

Calcul du taux de ralentissement de l'UC ρ dû au mécanisme de pagination :

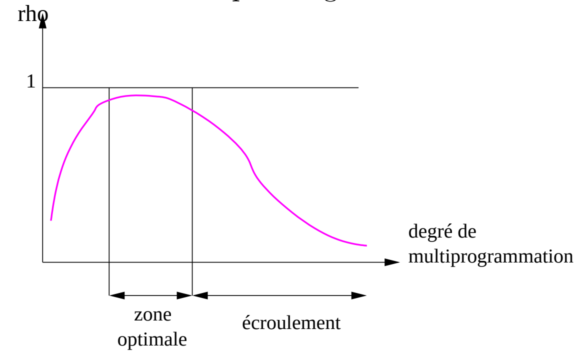
- Soient :
  - t le temps moyen d'exécution d'une instruction
  - p la probabilité d'un défaut de page
  - T le temps moyen de résolution d'un défaut de page ($T >> t$, facteur minimum de 10000)
- $\rho = \frac{t}{t+p.T} = \frac{1}{1+p.\frac{T}{t}}$
- Facteur ayant un impact important : $p.\frac{T}{t}$

  $\Longrightarrow$ Pour que ρ soit le plus proche possible de 1, il faut que p *soit le plus petit possible*

  $\Longrightarrow$ Il faut qu'un processus ait *assez de place pour loger son ensemble de travail*

Taux de ralentissement de l'UC ρ en fonction du degré de multiprogrammation :

- Degré faible : pas assez de processus à exécuter pendant les défauts de page
- Zone optimale
- Ecroulement : pas assez de mémoire pour loger les ensembles de travail



Solutions au phénomène d'écroulement

- Objectif : faire en sorte que chaque processus dispose d'assez de mémoire pour y loger son ensemble de travail
- Moyens :
  - Action sur *l'espace mémoire* alloué à chaque processus (objectif = allouer à chaque processus son espace de travail) et/ou
  - Action sur le *nombre de processus* se partageant la mémoire (objectif = éliminer les processus quand leur nombre est trop important)

Action sur l'espace mémoire Remplacement local ou global

Pages sur lesquelles s'applique l'algorithme de remplacement de pages :

- *Remplacement global* : choix effectué sur l'*ensemble* des pages physiques, quel qu'en soient les processus propriétaires
  - risque d'accaparation de la mémoire par un processus au détriment des autres
  - compétition peut empêcher tous les processus de s'exécuter dans de bonnes conditions
- *Remplacement local* : choix parmi les pages physiques possédées par le *processus faisant le défaut*

$\Longrightarrow$ Nécessité de contrôler la mémoire disponible pour chaque processus

- *Partition fixe* : nombre de pages divisé entre les processus
  - Peu adapté : évolution des besoins mémoire au cours du temps, manque d'équité
- *Partition variable* : re-calcul périodique de la taille de l'espace mémoire affecté à chaque processus

Action sur l'espace mémoire Evaluation espace de travail

- *Directe*
  - évaluation précise trop coûteuse
  - approximation avec les bits U et une horloge
- *Indirecte* : évaluation du taux de défaut de page par processus P
  - si taux < Dmin on enlève une page physique à P
  - si taux > Dmax on alloue une page physique supplémentaire à P

Contrôle de la charge par processus

Principe :

- On tente de conserver pour chaque processus son ensemble de travail en mémoire
- Si on n'y arrive pas, c'est qu'il y a trop de processus $\Longrightarrow$ on réquisitionne toutes les pages physiques possédées par un processus (le moins prioritaire par exemple)

Régulation globale de la charge

- Choix empirique d'un *indicateur de fonctionnement* du système, permettant de savoir si on est dans la zone optimale ou la zone d'écroulement
  - Exemples d'indicateurs : taux de défaut de page, temps moyen entre défauts de page, taux d'occupation du contrôleur disque
- Mesure régulière de cet indicateur
- Ajustement du degré de multiprogrammation pour maintenir le facteur dans une fourchette acceptable (réquisition de tout l'espace mémoire d'un processus)

5 Limitation de la consommation mémoire

5.1 Influence de la taille des pages

Impacts de l'augmentation de la taille des pages p sur la consommation mémoire du système :

- *Positifs*
  - *Diminition taille de la table des pages* : à espace virtuel de taille égale, moins de pages
  - *Diminution temps de transfert disque* : amortissement du temps de positionnement sur une piste
- *Négatifs*
  - *Fragmentation interne* : espace perdu venant du fait qu'un programme de fait pas un nombre entier de pages. En moyenne $p/2$ par région

Taille optimale pour limiter la consommation mémoire :

- Soient
  - p la taille d'une page
  - v la taille de l'espace virtuel

  - d la taille d'un descripteur de page virtuelle (la taille occupée par la table des pages est $d.\frac{v}{p}$)
  - on suppose une seule région par processus
- Place totale perdue par processus : $\frac{p}{2} + d.\frac{v}{p}$
- Quand p croit, cette fonction commence par décroître puis croit
- Taille optimum quand dérivée nulle, à savoir $p = \sqrt{2dv}$

Taille optimale pour limiter la consommation mémoire :

Exemple 21. — $d = 8$
- Optimum atteint pour une taille de page entre 2^{11} et 2^{12}

p	**place perdue**	**%place perdue**
256	16 512	3,1
512	8448	1,6
1024	4608	0,8
2048	3072	0,6
4096	3062	0,6
8192	4608	0,8

Remarques 22. — *La place perdue reste limitée par rapport à une gestion par zone (une demi-page par région)*
- *Le calcul précédent ne tient pas compte de l'amélioration des transferts disque avec des grosses pages*
- *Taille typique des pages de 512 octets à 8Ko*

5.2 Fonctions de pagination adaptées

Limitation de la consommation mémoire par utilisation de fonctions de pagination adaptées
- Constat : plus l'espace virtuel est grand, plus la table des pages est grande

Exemple 23. (taille de DPV de 32 bits, espace virtuel de 32 bits, tables linéaires)

taille des pages (octets)	taille table des pages (octets)	nombre pages table des pages
512	2^{25} (32 Mo)	2^{16} (65 536)
1024	2^{24} (16 Mo)	2^{14} (16 384)
2048	2^{23} (8 Mo)	2^{12} (4 096)
4096	2^{22} (4 Mo)	2^{10} (1 024)

Limitation de la consommation mémoire par utilisation de fonctions de pagination adaptées
- Table des pages d'un processus doit *résider en mémoire physique* (accédée par l'UGM)
- Adressage clairsemé (trous parmi les zones licites)
- Si on a un espace virtuel par processus, il y a une table des pages par processus
- Conserver en mémoire uniquement la table des pages du processus actif trop coûteux (rechargement lors des changements de contexte) $\Longrightarrow$ on laisse donc en mémoire physique les tables des pages des processus présents en mémoire

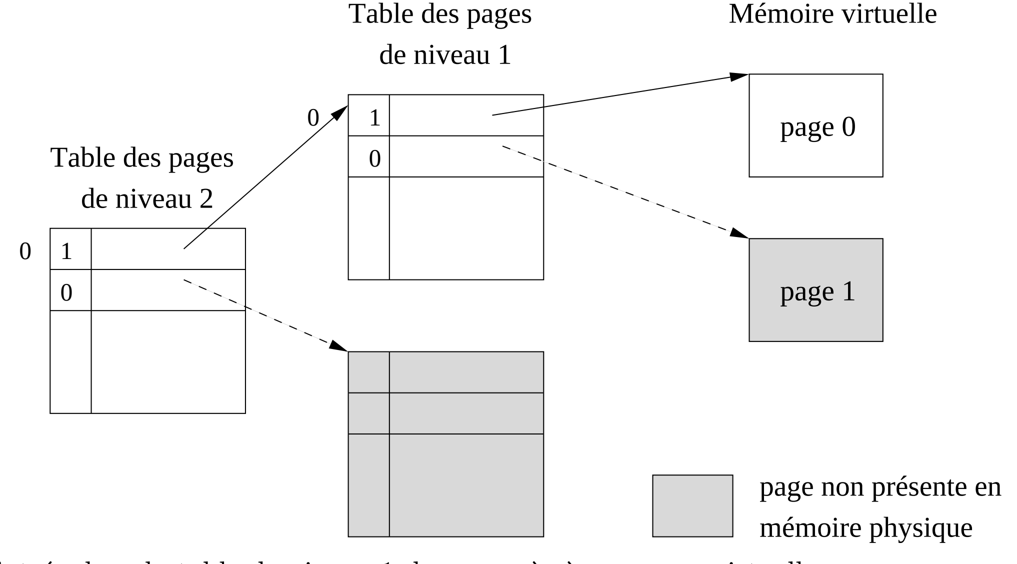
$\Longrightarrow$ Volume mémoire occupé par les tables est un réel problème
- Solutions possibles

— *Tables des pages à plusieurs niveaux* : découpage des tables en un arbre de tables, on ne conserve en mémoire que les niveaux utiles à un instant donné
— *Table des pages inverse* : on stocke les DPV dans la table des pages réelles

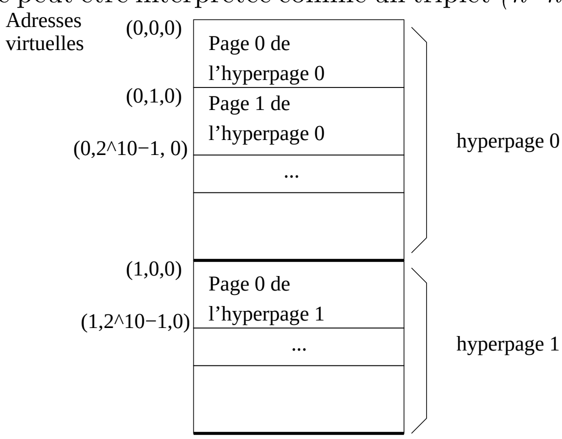
Tables des pages à plusieurs niveaux

Principe :

— Découpage de la table des pages en *pages*
— On conserve en mémoire uniquement les morceaux de tables (pages) *utiles* à un instant donné, uniquement pour les zones licites
— Détection d'une page de la table des pages manquante : *second niveau de pagination*, permettant de savoir si elle est présente, et si oui, à quelle adresse physique elle est implantée



— Entrée dans la table de niveau 1 donne accès à *une* page virtuelle
— Entrée dans la table de niveau 2 : donne accès à un *ensemble* de pages virtuelles contiguës (*hyperpage*, ou *livre*)
— Revient à considérer que la mémoire virtuelle est découpée en hyperpages, elles mêmes découpées en pages
— Une adresse virtuelle peut être interprétée comme un triplet *(n° hyperpage, n° page, déplacement)*



Tables des pages à plusieurs niveaux Contenu table des hyperpages (table de niveau 2)

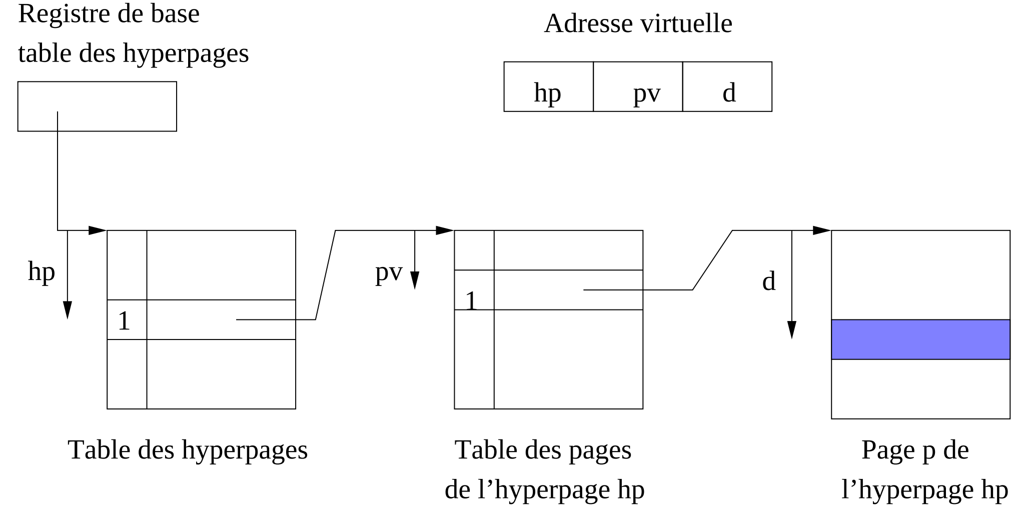
Chaque entrée *hp* contient :

— *présent (V)* : bit indiquant si la table des pages de l'hyperpage *hp* est présente en mémoire physique

— *adphys* : l'adresse physique de début de cette table des pages si elle est présente

Remarque 24. *Cette table des hyperpages joue le même rôle vis a vis de la table de pages, que la table de pages vis a vis de la mémoire*

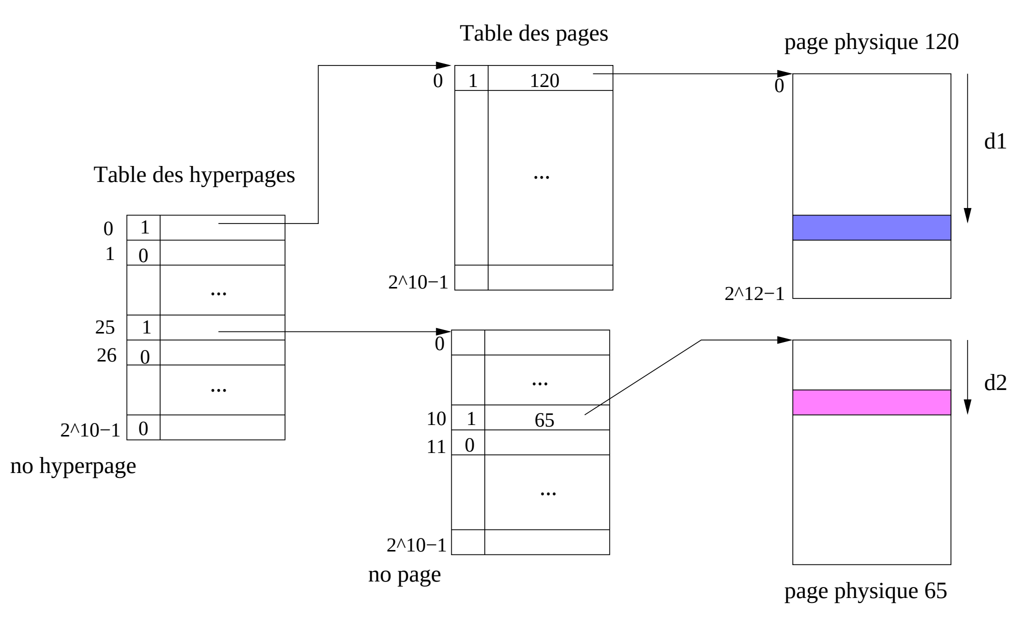
Tables des pages à plusieurs niveaux Interprétation d'une adresse virtuelle



Tables des pages à plusieurs niveaux Interprétation d'une adresse virtuelle *(hp, pv, d)*

— bit de présence de l'entrée hp de la table des hyperpages vaut 0 : *défaut d'hyperpage*

— bit de présence de l'entrée pv de la table des pages de l'hyperpage hp vaut 0 : *défaut de page*



Exemple 25. Exemple précédent, avec hyperpages de 2^{10} pages et pages de 2^{12} pages
- accès à *(0, 0, d1)* : fournit l'adresse physique *(120, d1)* (emplacement bleu)
- accès à *(25, 10, d2)* : fournit l'adresse physique *(65,d2)* (emplacement rose)
- accès à *(25, 11, d3)* : provoque un défaut de page
- accès à *(26, 50, d4)* : provoque un défaut d'hyperpage

Tables des pages à plusieurs niveaux Traitement d'un défaut d'hyperpage

1. Vérification du caractère licite de l'adresse
2. Recherche d'une page physique libre *pp* pour la table des pages manquante
3. Initialisation de cette table des pages.
   - En général, tous les bits de présence sont à faux. Il faut éventuellement transférer vers la page physique *pp*, l'image disque de la table des pages manquante.
4. Ré-exécuter l'instruction, qui va probablement provoquer un défaut de page

Tables des pages à plusieurs niveaux Volume occupé par les tables

Remarques 26. *Si on utilise tout l'espace virtuel, avec pages de 4Ko :*
- *pagination à un niveau :* $2^{20} * 2^2 = 2^{22}$ *octets de tables (4Mo)*
- *pagination à deux niveaux : il faut en plus* 2^{12} *octets (table des hyperpages), mais seuls ces 4 Ko doivent résider en permanence en mémoire physique, les autres sont soumis au va-et-vient*

Si on n'utilise qu'une partie de l'espace virtuel :
- *la pagination à deux niveaux permet de ne décrire complètement que la partie utile de cet espace (exemple : marqueur dans la table des hyperpages pour les zones illicites)*

Tables des pages inverse

- Volume des tables "directes" (linéaires, à plusieurs niveaux) proportionnelles à la taille des espaces virtuels adressables

— Incompatible avec les architectures à *grands espaces d'adressage* (64 bits)
— *Table des pages inverse* : on stocke les informations de traduction d'adresse dans la *table des pages physiques*

Tables des pages inverse Structures de données pour une page physique p

— identification de l'espace virtuel auquel elle appartient (numéro de processus s'il y a un espace virtuel par processus), ou marqueur si page disponible
— numéro de la page virtuelle dans cet espace virtuel

Tables des pages inverse Structures de données

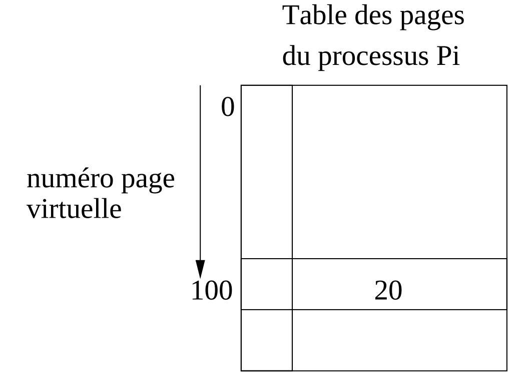


Table des pages directe

Table des pages inverse
du système

numéro page
physique

	Pi	100

0
20

Table des pages inverse

Tables des pages inverse Traduction d'adresse

1. Cache de traduction
2. Si absent, recherche dans la table des pages réelles d'une entrée *(Pi,pv)*
   — on ne travaille plus par indexation
   — accélération des accès : techniques de dispersion (hachage)
3. Si absent de la table des pages en mémoire, défaut de page, que l'on résout comme d'habitude (sauf identification des adresses disque)

Fonctions de pagination

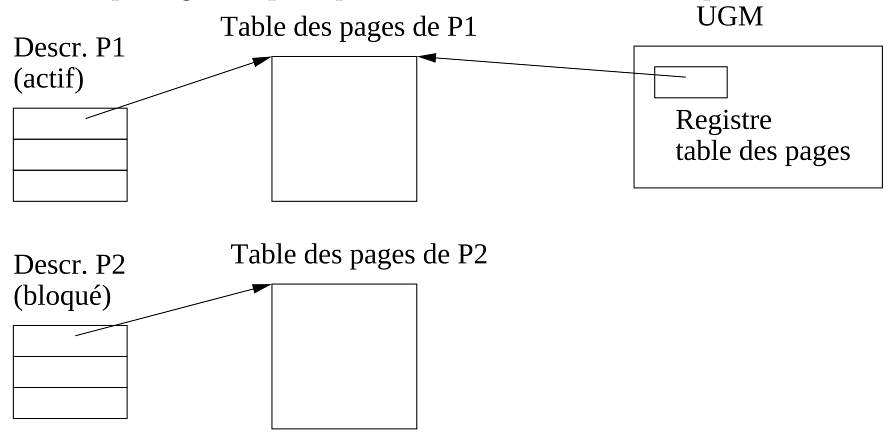
— Tables des pages directes
  — à un seul niveau (linéaires)
  — à plusieurs niveaux (hiérarchiques)
— Tables des pages inverses
— Caches de traduction
— Variations non étudiées
  — Pagination automatique des tables des pages en les mettant dans l'espace virtuel (superviseur)

6 Gestion mémoire et gestion du processeur

— *Objectifs* : examiner les liens entre gestion mémoire et gestion processeur : lien entre unité d'exécution (processus, thread) et espaces virtuels

Modèle d'exécution Un processus par espace virtuel

— Lien *un à un* entre unité d'exécution et espace virtuel
— On parle de processus *lourd*
— *Protection* des processus les uns par rapports aux autres (accès mémoire incorrects intentionnellement ou non)
— Mise en œuvre
  — Registre contenant l'adresse de la table des pages courantes dans l'UGM
  — Sauvegarde de ce registre lors des changements de contexte
  — Vidage du TLB lors des changements de contexte (sauf champ ASID)
  $\Longrightarrow$ Changements de contexte plus longs que sans pagination (rechargement TLB, registre de plus à sauvegarder)
  $\Longrightarrow$ Mémoire partagée ne peut pas être utilisée directement pour communiquer



Modèle d'exécution Plusieurs processus se partageant le même espace virtuel

— Deux notions différentes :
  — *Processus léger* (thread)
  — *Tâche* : comprend un espace virtuel et un ensemble de thread
— Pas de protection entre threads de la même tâche, protection entre threads de tâches différentes ($\Longrightarrow$ change les moyens de communication entre threads), partage de mémoire par construction
— Mise en œuvre
  — Pas de contexte mémoire (registre de début de table des pages) dans un thread $\Longrightarrow$ changement de contexte entre threads léger
  — Changement de contexte entre threads de tâches différentes plus lourd (sauvegarde/restauration du contexte mémoire, vidage du TLB)

Modèle d'exécution Un espace virtuel pour tous les processus

— Utilisé dans les architectures à *grands espaces d'adressage* (64 bits)

— Jamais de changement d'espace virtuel $\Longrightarrow$ partage des objets en mémoire très simple
— En l'absence de segmentation, protection des objets en mémoire difficile à assurer

7 Autres mécanismes permettant le va-et-vient

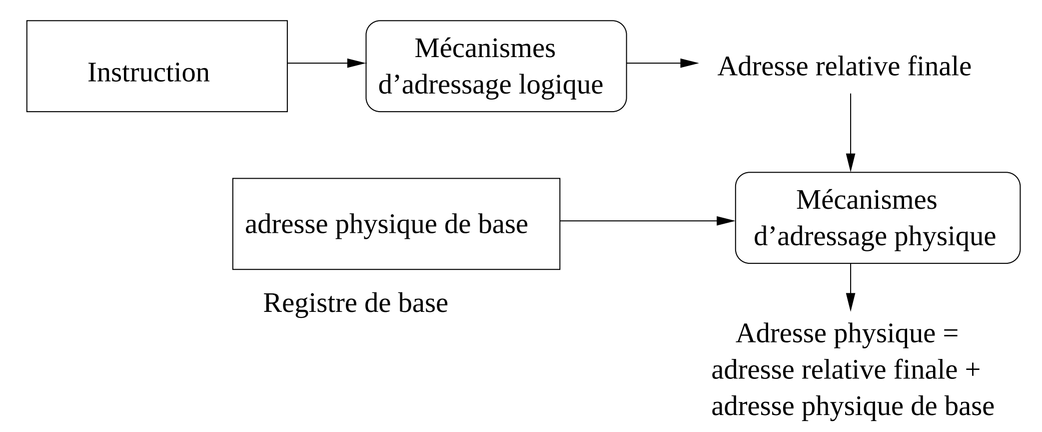
7.1 Adressage par registre de base

Adressage par registres de base

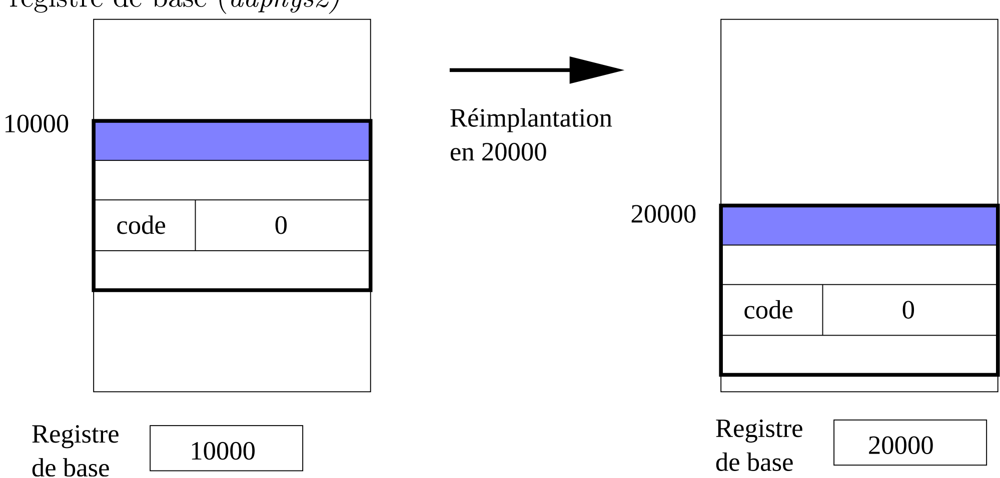
Definition 27 (Adressage par registre de base (sans pagniation))**.** — Toutes les adresses figurant dans les instructions, ou manipulées par les instructions, sont des adresses *relatives*

— Les mécanismes d'adressage logique (indirection, indexation,...) *produisent une adresse relative* (adresse relative finale)

— Seul le registre de base contient une *adresse physique* qui est ajoutée à l'adresse relative finale pour produire l'adresse physique



— Déplacement programme en cours d'exécution de *adphys1 à* adphys2 : modification contenu du registre de base (*adphys2)*



Remarques 28. — *On peut avoir plusieurs registres de base, comme sur le 8086 (CS, DS, SS, ES)*

- *L'existence de registres de base ne garantit pas pour autant que tout programme peut être ré-implanté, il faut les utiliser correctement (exemple ci-dessous avec une réimplantation entre (1) et (2))*
- *Permet la réimplantation dynamique mais pas de va-et-vient par bloc (va-et-vient global seulement)*

  X RW 1
  MOV X,DS (1)
  ...
  MOV DS,X (2)

7.2 Adressage segmenté

Adressage segmenté (sans pagination)

- Variation de d'adressage par registre de base
- *Segment* = unité de structuration, partage et protection de l'information
- *Descripteur de segment* : contient les informations de taille, protection, et l'*adresse physique* d'implantation du segment
- *Réimplantation dynamique* : modification du descripteur de segment (changement adresse d'implantation)

7.3 Segmentation et pagination

- *Pagination* : facilite l'implantation des programmes en mémoire physique d'un espace virtuel linéaire
- *Segmentation* : offre à l'utilisateur un espace virtuel composé de plusieurs espaces linéaires *indépendants* (résout les problèmes de partage, protection, gestion des données de taille variable)

$\Longrightarrow$ Ces mécanismes sont *complémentaires* et peuvent être utilisés de manière *conjointe*

- Manières de combiner segmentation et pagination :
  - Paginer chaque segment
  - Implantation des segments dans un grand espace linéaire, que l'on pagine ensuite

Paginer les segments

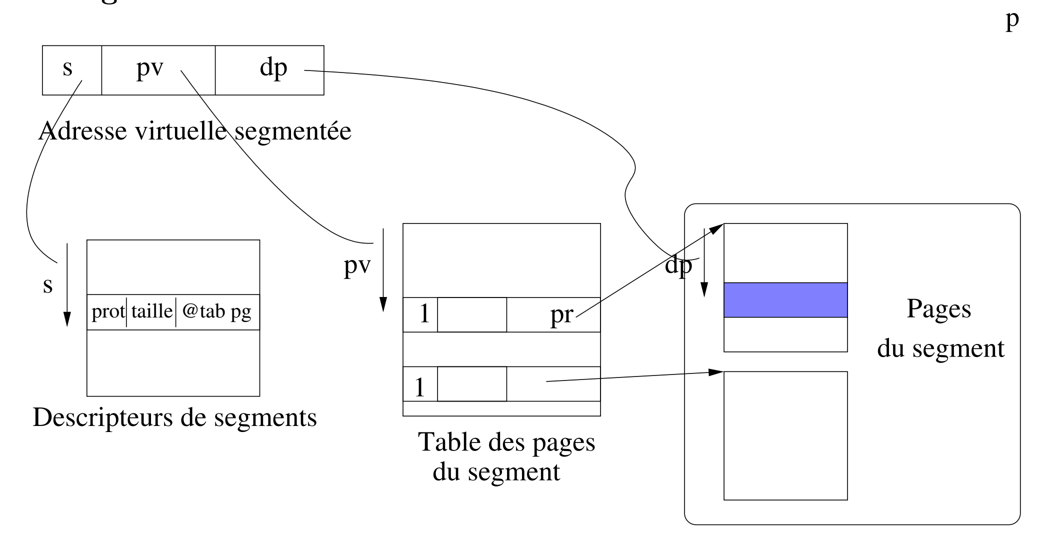
Principe :

- Chaque segment est un espace linéaire que l'on pagine

— Une table des pages (ou hiérarchie de tables) par segment
— Descripteur de segment contient (hors taille + droits) l'adresse physique de la table des pages du segment
— Adresse virtuelle = (nom_segment, déplacement_segment)
— déplacement_segment interprété comme un couple (numéro_page, déplacement_page)

Paginer les segments

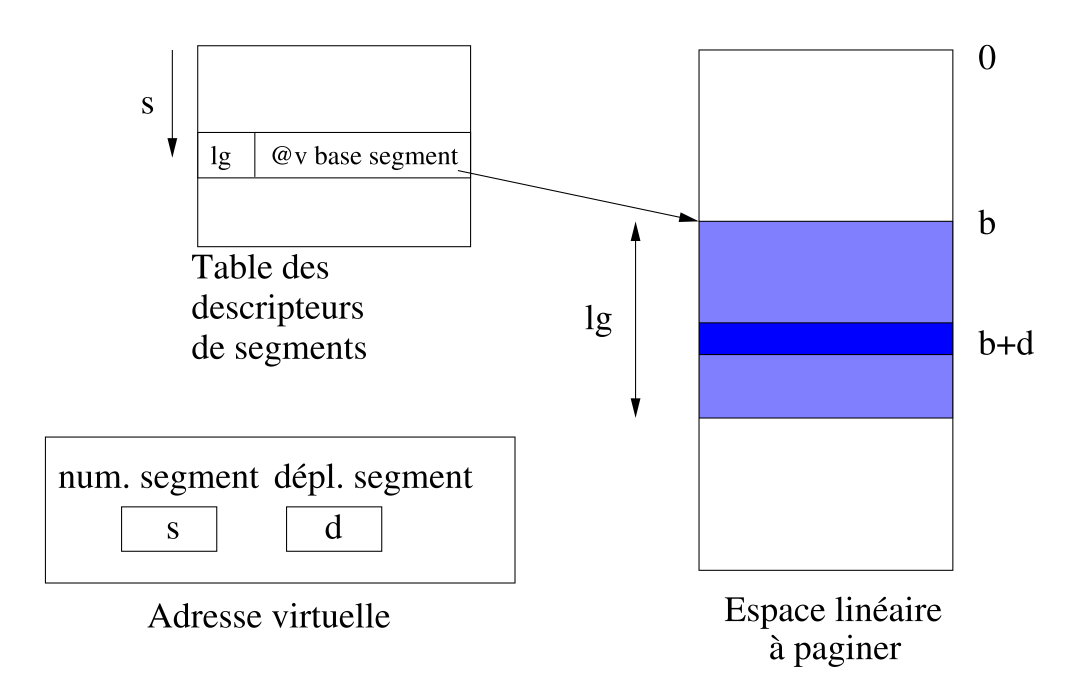
p



Remarques 29. — *Force chaque segment a avoir sa propre table des pages (inefficace pour les petits segments)*
— *La segmentation rend inutile l'allocation de la table des pages pour la taille adressable maximale d'un segment*
— *Une adresse de numéro de segment s ne permet d'accéder qu'à s $\Longrightarrow$ bien que l'on ait deux niveaux de tables, il ne s'agit pas d'une pagination à deux niveaux*
— *Partage d'un segment possible en partageant sa table des pages*

Paginer l'espace où sont implantés les segments

Principe :
— Implantation des segments dans un espace linéaire
— Pagination de cet espace linéaire



Remarques 30.
- *Il faut résoudre le problème d'implantation d'un segment dans l'espace linéaire (cf. gestion mémoire par zones)*
- *Plus de table des pages par segment*
- *Deux segments différents peuvent être situés dans la même page et se partagent alors le même DPV* $\Longrightarrow$ *intéressant si on a beaucoup de petits segments*

Troisième partie

Allocation de la mémoire par zone

Allocation dynamique de mémoire

Objectif :

- Demande de mémoire supplémentaire *à l'exécution*
- Tailles et durées d'utilisation des zones de mémoire *quelconques*

Interface typique :

- void *malloc(size_t size) : demande d'une zone de mémoire de taille *size* et retour de son adresse
- void free(void *ptr) : libération d'une zone de mémoire allouée au préalable (rq : on ne passe pas la taille en paramètre)

Domaines d'utilisation :

- *Systèmes sans pagination* : allocation de mémoire *réelle*
- *Systèmes avec pagination* : allocation de zones dans l'espace d'adressage *virtuel* utilisateur, allocation en mémoire *physique* pour le système d'exploitation

Allocation dynamique de mémoire

Terminologie

- *Zone* : suite d'emplacements mémoire contigus, de taille non fixée a priori
- Zone caractérisée par son adresse de début et sa taille
- *Zone libre* (*trou*) : zone de mémoire non allouée par le système
- *Zone occupée* : partie de mémoire allouée à un processus

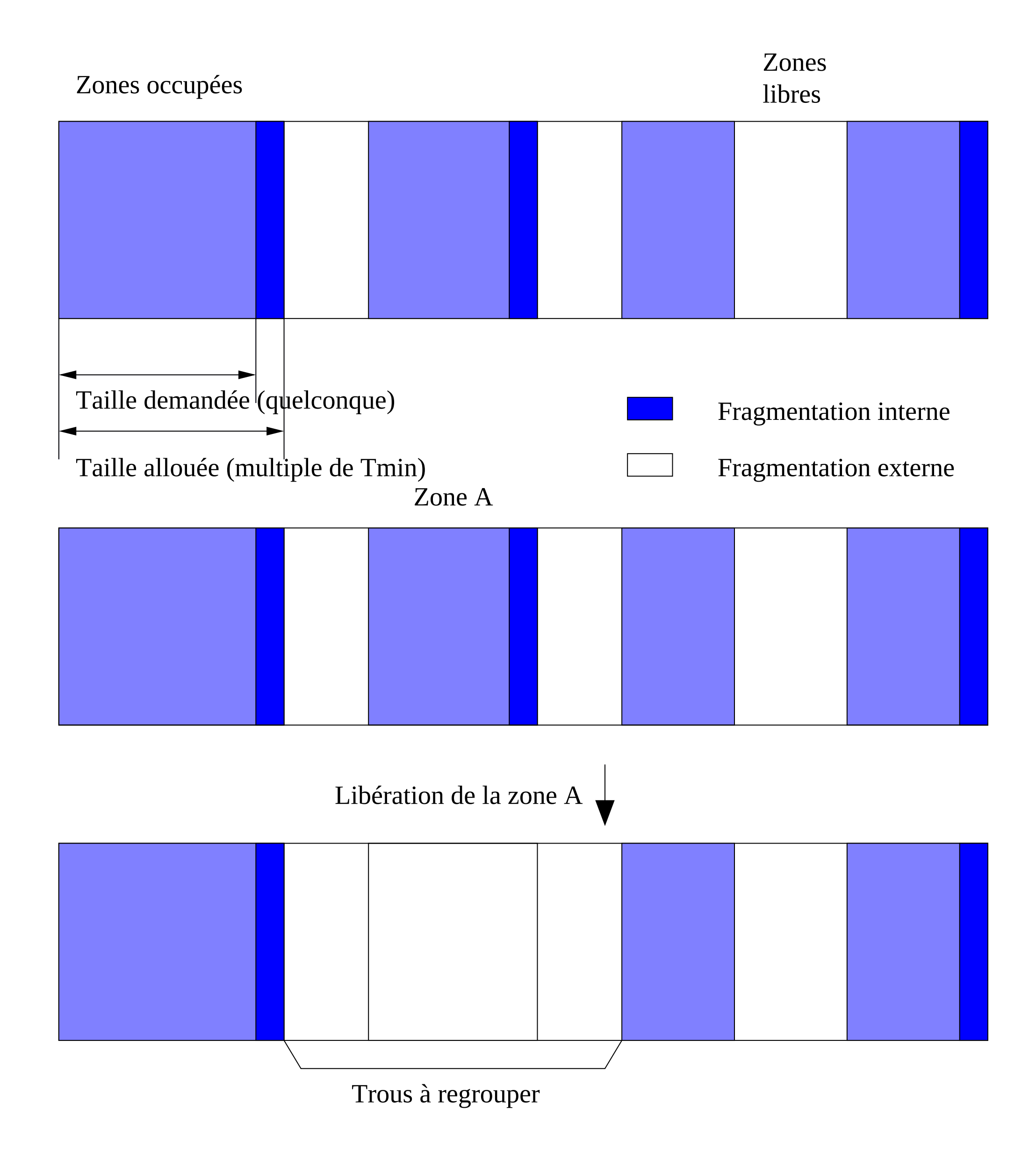
1 Problèmes à résoudre

Problèmes à résoudre

- Distinction entre zones libres et zones occupées
  $\Longrightarrow$ Structure de données adaptée
- *Allocation* : parcours de la structure de données pour trouver une zone libre
- *Libération* : réintégration du bloc dans la structure de données

Fragmentation

- Fragmentation *externe*
  - Au fil des allocations/libérations, l'espace mémoire est constitué d'un mélange de zones libres et occupées
  - Fusion de trous adjacents en mémoire lors de la libération
  - La place prise par les zones libres peut être perdue si les zones libres sont de trop petite taille
- Fragmentation *interne* :
  - Taille allouée $\geq$ taille demandée (multiple d'une taille minimum de bloc T_{min}, ou autres contraintes sur tailles de blocs)
  - Motivation : limitation taille de structures de données
  - Conséquence : place perdue (taille_allouée - taille_demandée)



2 Algorithmes d'allocation dynamique

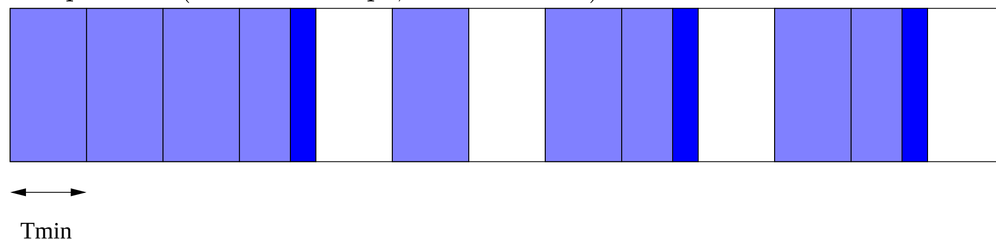
2.1 Classes d'algorithmes d'allocation dynamique

Classes d'algorithmes d'allocation dynamique

- *Bitmap* : table de bits (1 bit par bloc)
- *Sequential fits* : structure de liste stockée dans les trous
- *Indexed fits* : autre structure de données (e.g. arbre) stockée dans les trous
- *Buddy systems*
- Politiques *hybrides* : dépendante de la taille de bloc demandée

2.2 Bitmap

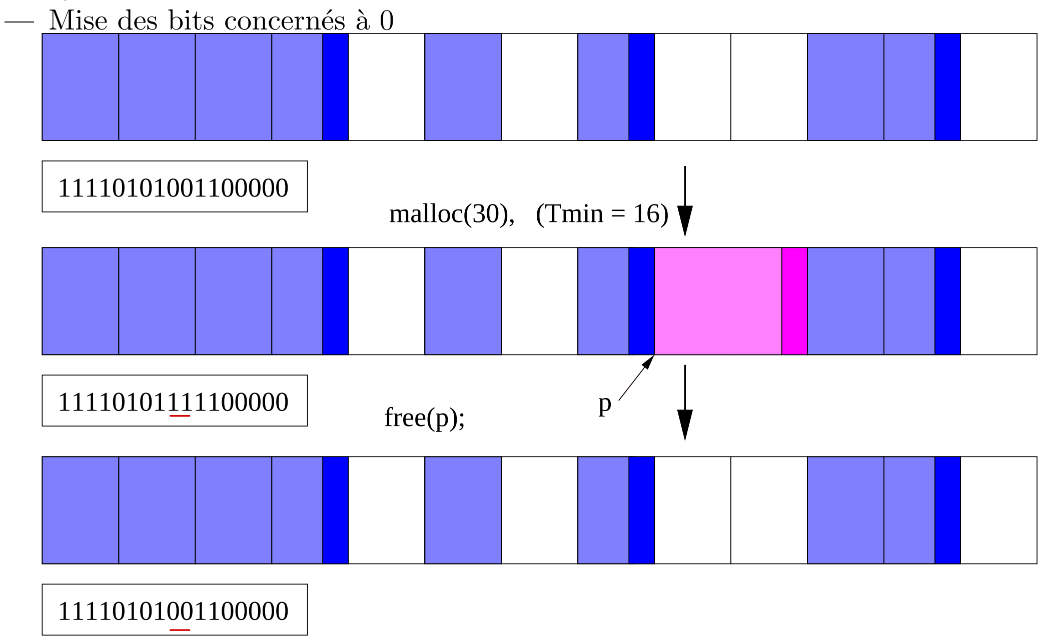
- Allocation par multiple de bloc de taille fixée T_{min}
- Un bit par bloc (1 = bloc occupé, 0 = bloc libre)



1111010110110000000

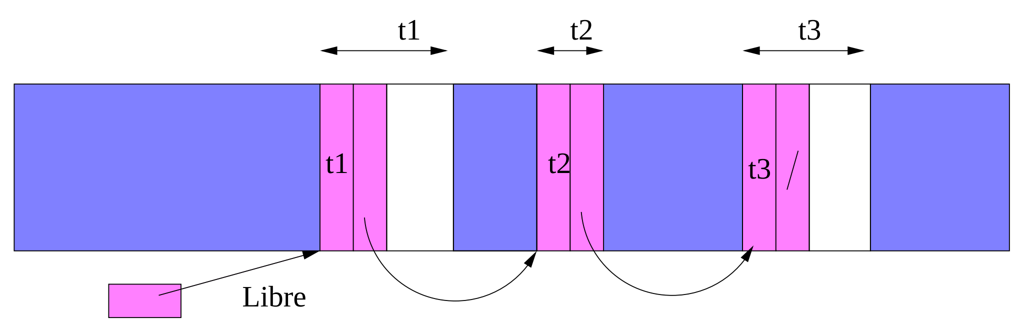
Bitmap

- Allocation
  - Arrondir la taille demandée au T_{min} supérieur $\Longrightarrow$ taille allouée = $n * T_{min}$
  - Recherche de n blocs consécutifs à 0, puis mise à 1
- Libération
  - Vérification dans la bitmap que la libération correspond bien à une zone allouée (bits à 1)
  - Mise des bits concernés à 0



2.3 Sequential fits

- Chaînage des trous dans une *liste*
- Mémorisation de la structure de liste *dans les trous*

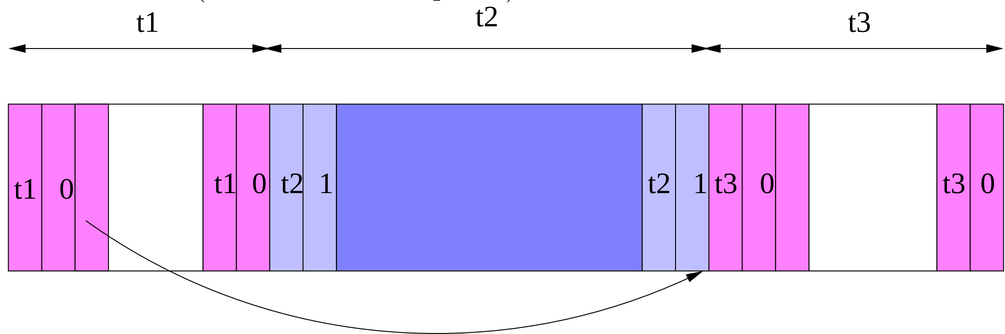


— Allocation : parcours de la liste des blocs libres
— Libération : insertion dans liste des blocs libres (+ fusion avec blocs adjacents si applicable)

Organisation de la liste :
— Par *adresse croissantes* : facilite le regroupement des zones en cas de libération
— Par *taille croissante* : facilite la recherche d'un bloc d'une taille donnée

Sequential fits : technique pour la fusion

Boundary tags : pour tout bloc (libre ou occupé)
— *Entête* (*header*) et *prologue* (*footer*) contenant :
  — la taille du bloc
  — l'état du bloc (libre - 0 - ou occupé - 1)



Sequential fits

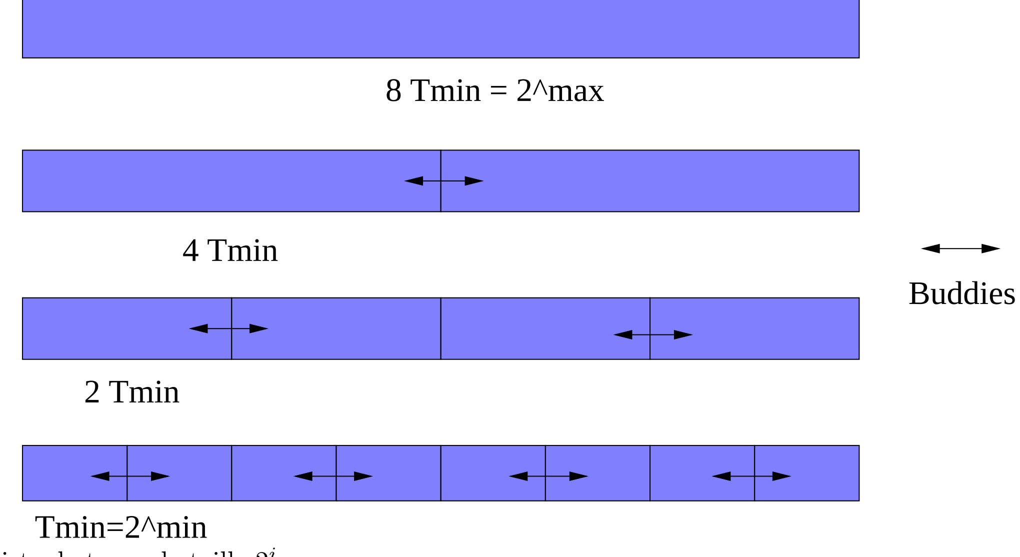
Stratégies courantes de recherche d'un bloc :
— *First fit* : liste des trous triée par adresse, recherche du premier trou de la liste de taille ≥ à la taille demandée
— *Next fit* : variation du first fit ou on gère la file circulairement en repartant lors de la recherche de la dernière zone allouée
— *Best fit* : on recherche la plus petite zone convenable (paradoxalement, mauvaise utilisation de la mémoire due à une multiplicité de petits trous - résidus)

2.4 Indexed fits

Structure de données élaborée pour mémoriser les blocs libres :
— *Arbre binaire équilibré* permettant de trier les blocs par taille
— *Arbre cartésien* trié à la fois selon la taille des trous et leur adresse
— Stockée dans les trous eux mêmes
— Segregated fits : structure de données et algorithme d'allocation différent par taille de bloc

2.5 Buddy systems

- On n'alloue que certaines tailles de blocs
  - *Binary buddy* : puissances de deux
  - *Fibonacci buddy* : taille membres d'une suite de Fibonacci
- Chaque bloc a son bloc compagnon (*buddy*) adjacent qui est le seul bloc avec qui il peut être fusionné en cas de libération
- Gros taux de fragmentation interne à cause des choix de tailles de blocs



- Liste de trous de taille 2^i
- Initialement, listes vides sauf 2^{max}

```
char *allouer(int T) {
    calcul de i tel que 2^{i-1} < T ≤ 2^i
    adr=trouver_trou(2^i);
    return (adr);
}
```

```
char *trouver_trou ( 2^i ) {
    if (i > max) return -1;
    if (liste(i) vide) {
        ad=trouver_trou(2^{i+1});
        if (ad != -1) {
            diviser ce trou en 2 trous de taille 2^i
            placer ces 2 trous 2^i dans la liste(i)
        } else return -1;
    }
    adresse_trou = extraire_1er_trou_liste(i);
    retour adresse_trou;
}
```

3 Ramasse miettes

Danger de la libération manuelle de mémoire (free)

— Oubli de libération
— Double libération
— Utilisation d'une zone après libération

$\Longrightarrow$ Libération automatique de la mémoire (Ramasse-miettes, Garbage Collection)

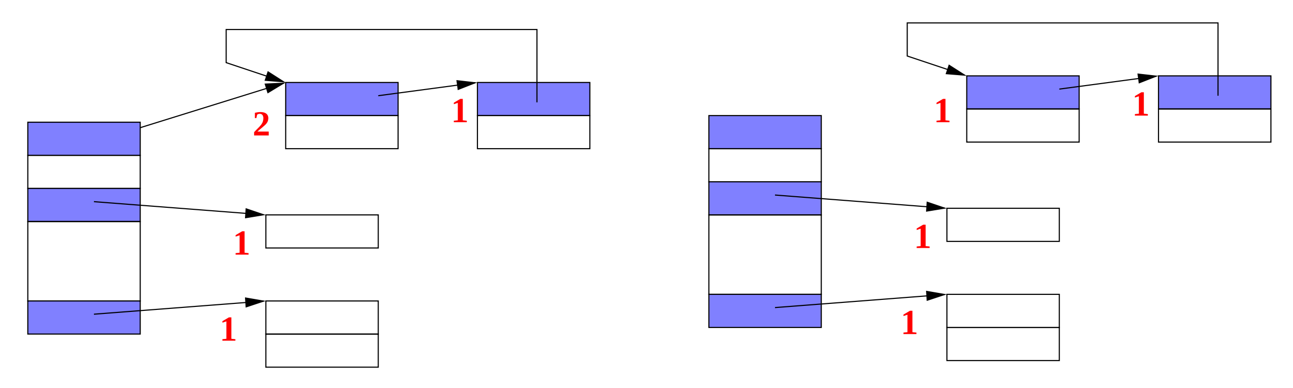
— Objet *racine* : utile par définition (ex : pile)
— Objets utiles : accessibles directement ou indirectement à partir de l'objet racine via une chaîne de références

Remarque 31. *Nécessite de distinguer les références des données simples dans les objets*

Comptage de références (Reference Counting)

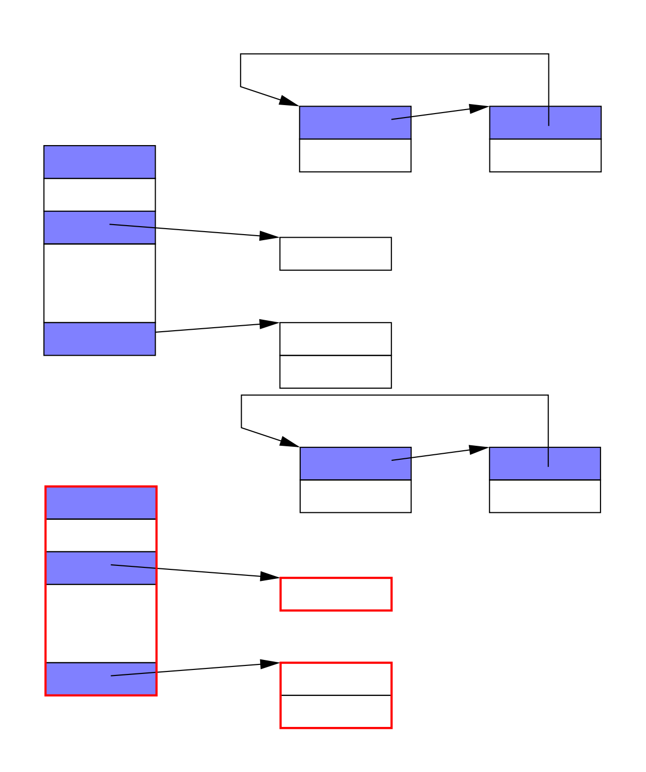
— Compteur de références par objet
  — Ajout d'une référence : incrémentation du compteur
  — Retrait d'une référence : décrémentation du compteur
  — Destruction de l'objet quand son compteur de références atteint 0
— Utilisé dans les SGF pour la destruction des fichiers (liens physiques)

Remarque 32. *Ne libère pas les structures cycliques*



Marquage et balayage (mark and sweep)

— Marquage
  — Marquage des objets racines
  — Marquage de tout objet non marqué référencé par un objet marqué
  — (Parcours du graphe des références)
— Balayage : libération de la mémoire de tout objet non marqué



Quatrième partie

Liaison et partage des objets dans un programme

1 Partage d'objets

1.1 Définitions et motivations

Définitions

Definition 33 (Partager). — Sens commun : *"Posséder avec d'autres", "mettre en commun"* (petit Larousse)
— Sens informatique : ne pas dupliquer de l'information utile à plusieurs processus (disque, mémoire)

Motivations

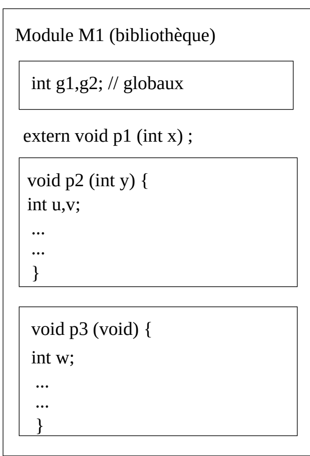
— Interfaces utilisateur (graphique, son, bibliothèques d'exécution de langages)
  $\Longrightarrow$ Mise du code dans des *bibliothèques* volumineuses
  $\Longrightarrow$ Intégration des bibliothèques dans les exécutables (disque) de moins en moins raisonnable
— *Code exécutable* potentiellement partagé entre plusieurs processus
  $\Longrightarrow$ Duplication de ce code en mémoire inutile

Objets partagés

— *Modules* (typiquement *bibliothèques*)
— *Objets* (au sens de la programmation à objets)

Modules

— Procédures
— Variables locales
  — durée de vie de la procédure
  — 1 copie par appel en cours
— Paramètres formels
— Variables globales
  — durée de vie $\geq$ procédure
  — en général, en un seul exemplaire
— Objets externes : définis à l'extérieur du module



1.2 Propriétés attendues d'un mécanisme de partage

- Connaissance de l'*interface* du module uniquement
- Pas de connaissance :
  - de la mise en œuvre du module (variables et procédures internes, utilisation d'autres modules)
  - de son utilisation par d'autres processus (adresse d'implantation)

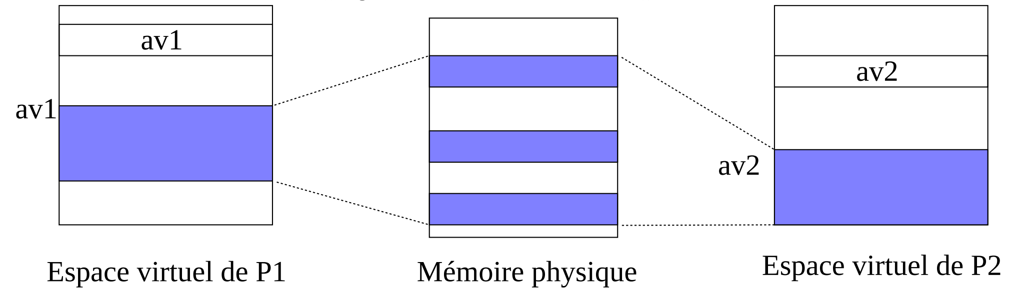
1.3 Partage dans un espace paginé

Cadre

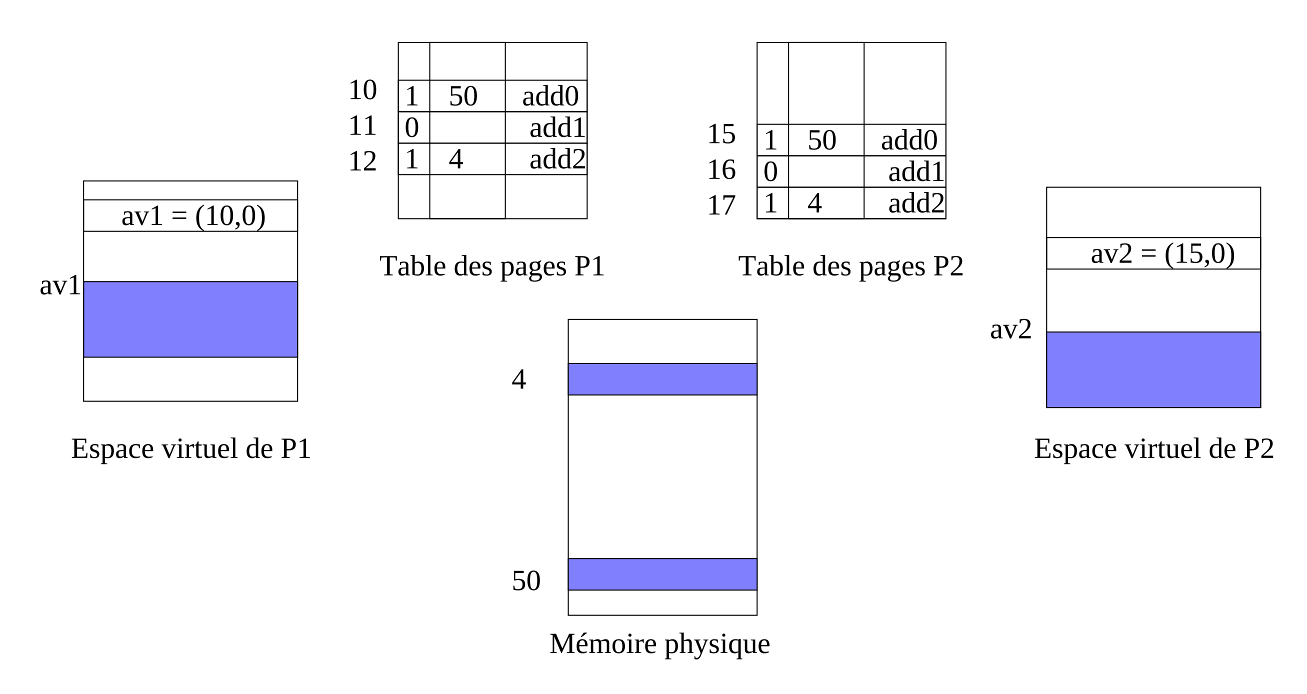
- Processus dotés d'espaces virtuels linéaires paginés
- Espaces d'adressages privés (une table des pages par processus)
- Objet à partager : objet O formé de pages contiguës (*région*)
  - Région de code
  - Bibliothèque

Mécanisme de partage

- Implantation de O dans les espaces virtuels des processus le partageant
- Les adresses d'implantation peuvent être *différentes*
- Les contenus des tables de pages doivent être *identiques*



Mécanisme de partage Mise en œuvre (table des pages linéaires)



Remarques 34. — *Objets à partager ont une taille multiple de la taille d'une page*

- *Adresse(s) de l'objet partagé sur une frontière de page*
- *Deux DPV référencent la même page réelle*
  - *Impact sur l'algorithme de remplacement de page*
  - *Duplication inutile de l'information information contenue dans les DPV*

Mécanisme de partage Mise en œuvre (table des pages hiérarchiques)

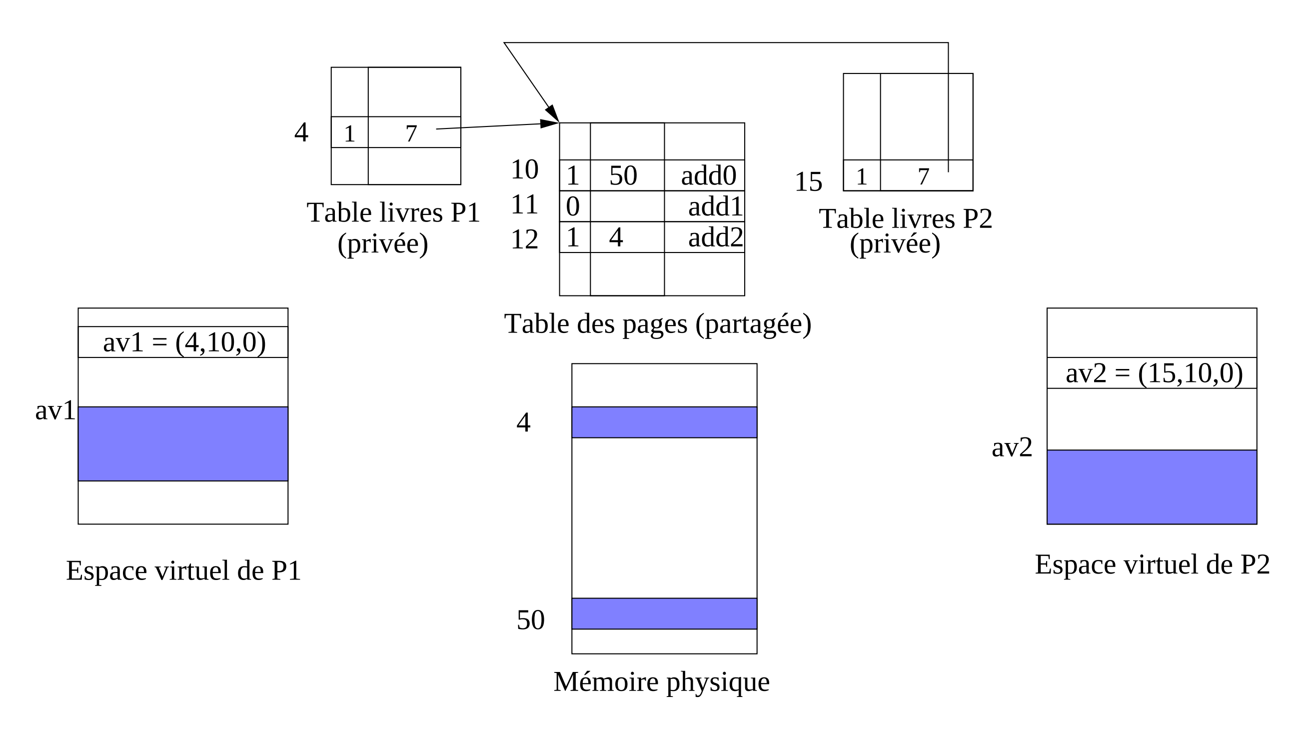
Principe :

— Partage non seulement des pages en mémoire, mais des DPV les décrivant

Mise en œuvre :

— Table des hyperpages (livres) *privée* à chaque processus

— Pour la région partagée, pages des tables des pages *partagées* par les processus se partageant l'objet



Remarques 35. — *Adresses des objets à partager sur des frontières d'hyperpages (livres)*

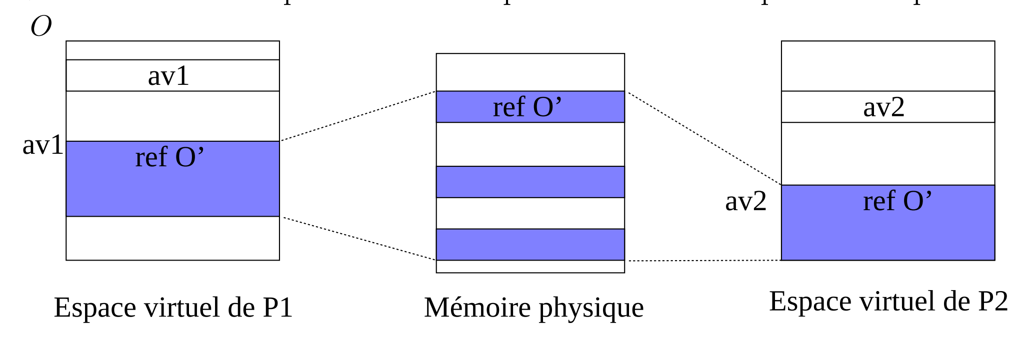
— *Un seul DPV par page réelle*

$\Longrightarrow$ *Pas d'impact sur l'algorithme de remplacement de page*

Mécanisme de partage Partage d'un objet contenant une référence

— Si O est partagé, la référence à O' l'est également

$\Longrightarrow$ La référence doit pouvoir être interprétée correctement par tous les processus utilisant O

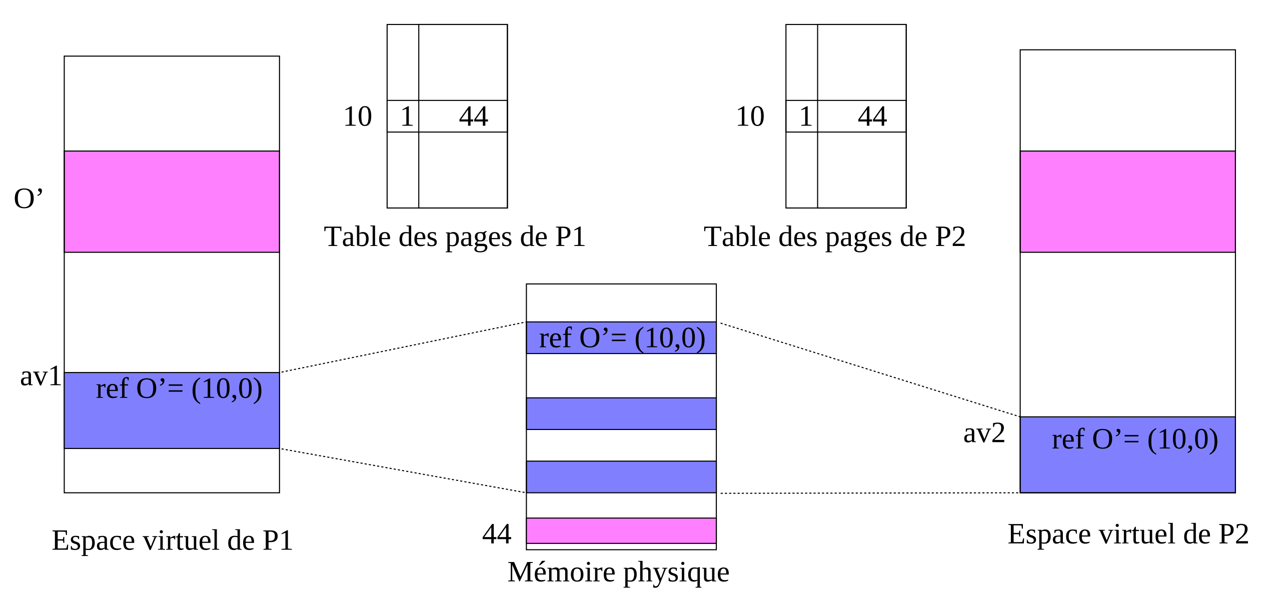


Cas à considérer :

— Référence à O' est une adresse virtuelle directement utilisée par le mécanisme de pagination (adressage *direct*)

— Référence à O' utilise un adressage *calculé* qui ne fournira l'adresse virtuelle finale qu'à l'exécution (ex : adressage basé)

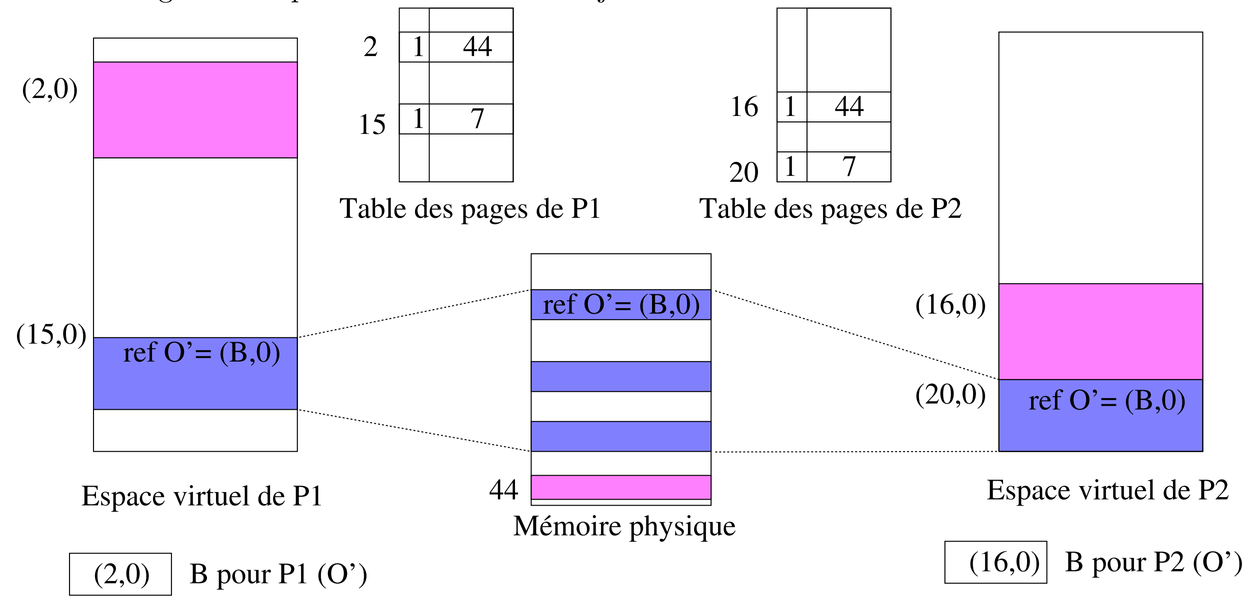
Adressage direct pour références entre objets :



Adressage direct pour références entre objets :

- Interprétation correcte $\Longrightarrow$ O' doit être à la même adresse virtuelle dans tous les processus
  $\Longrightarrow$ Partage *non modulaire* : le partage de l'objet O nécessite de connaître les objets utilisés par O
- N'est pas un mécanisme de partage *général*
- Utilisable dans des *cas particuliers* : partage de code entre processus exécutant le même code

Adressage calculé pour références entre objets :



Adressage calculé pour références entre objets :

- Pas d'adresses virtuelles dans le code des programmes $\Longrightarrow$ pas de contrainte sur le placement des objets partagés
- Contenu du registre de base différent par processus se partageant l'objet
- Partage modulaire, si registre de base différent par objet partagé

Segments de mémoire partagés UNIX

Interface :

— Création : *int shmget(key_t key, int size, int shmflg) ;*
— Attachement : *void *shmat(int shmid, void *shmaddr, int shmflg) ;*
— Détachement : *int shmdt(void *shmaddr) ;*

Propriétés :

— Partage des données contenues dans le segment sans interprétation par le système
— Deux processus peuvent voir le segment à deux adresses virtuelles différentes

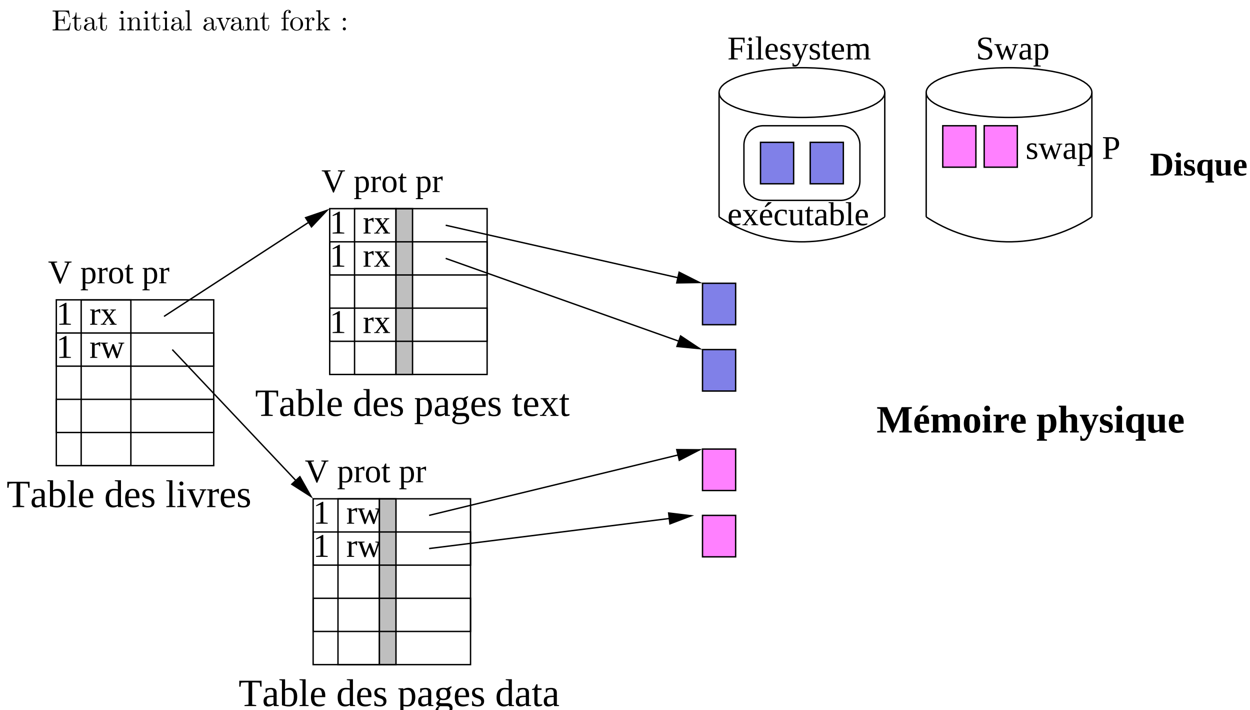
1.4 Etude de cas : partage et fork Unix

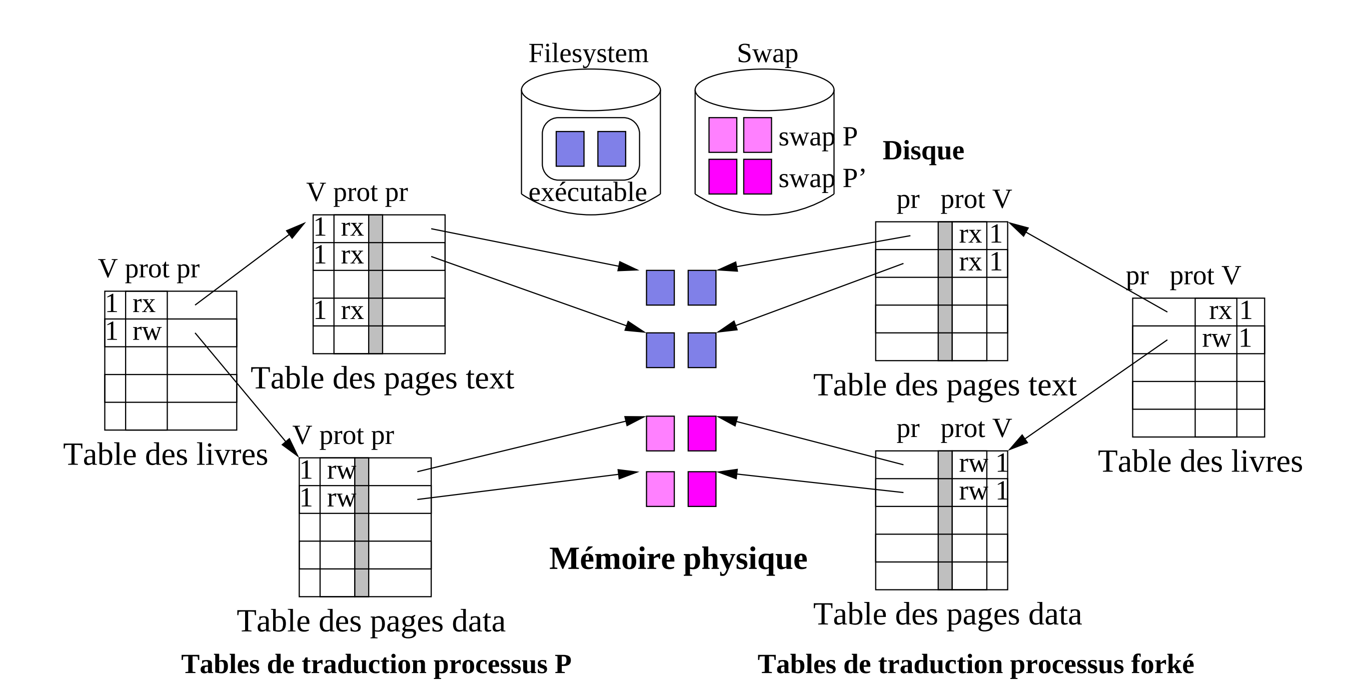
Etude de cas : partage et fork Unix

— Interface : *pit_t fork(void) ;*
— Sémantique : duplication de l'espace d'adressage du processus appelant (code, data, pile), partage des fichiers ouverts
— Objectif : mise en œuvre efficace du fork
— Hypothèses :
  — Table des pages hiérarchiques
  — Swap allouées au chargement

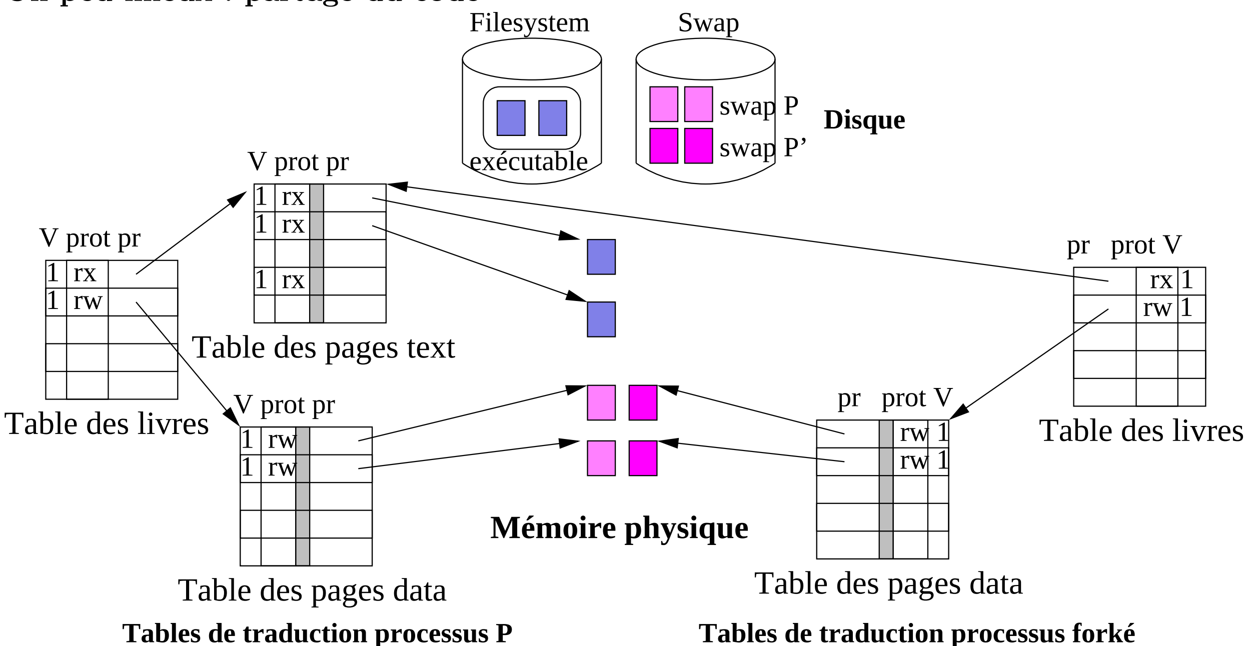
Réalisation basique

Etat initial avant fork :





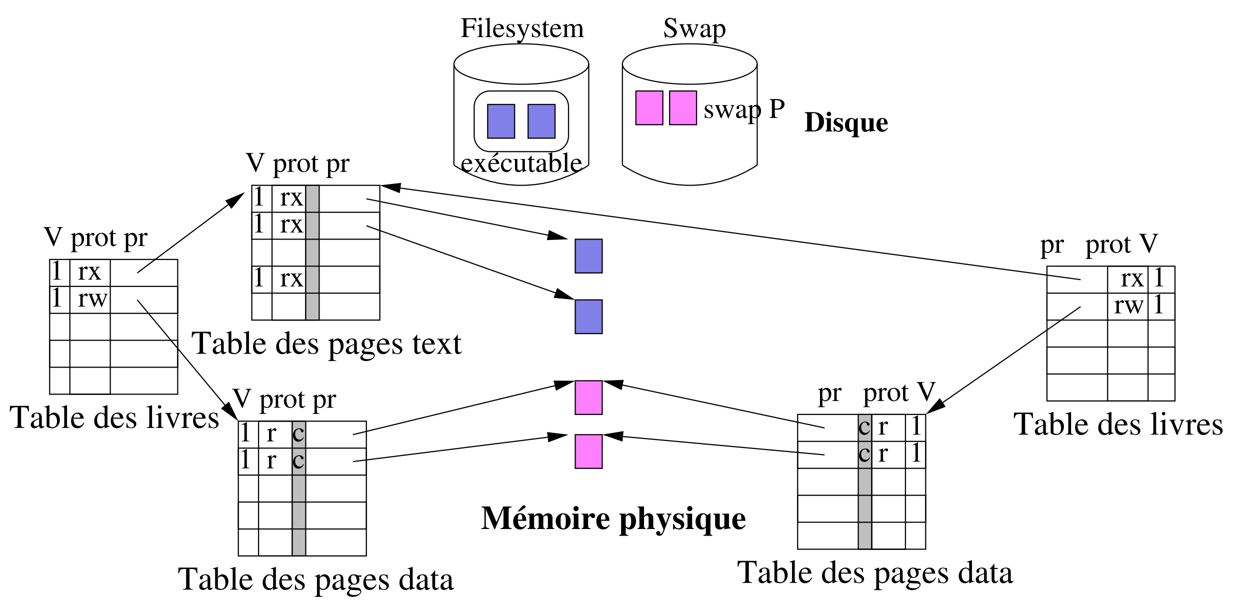
Un peu mieux : partage du code



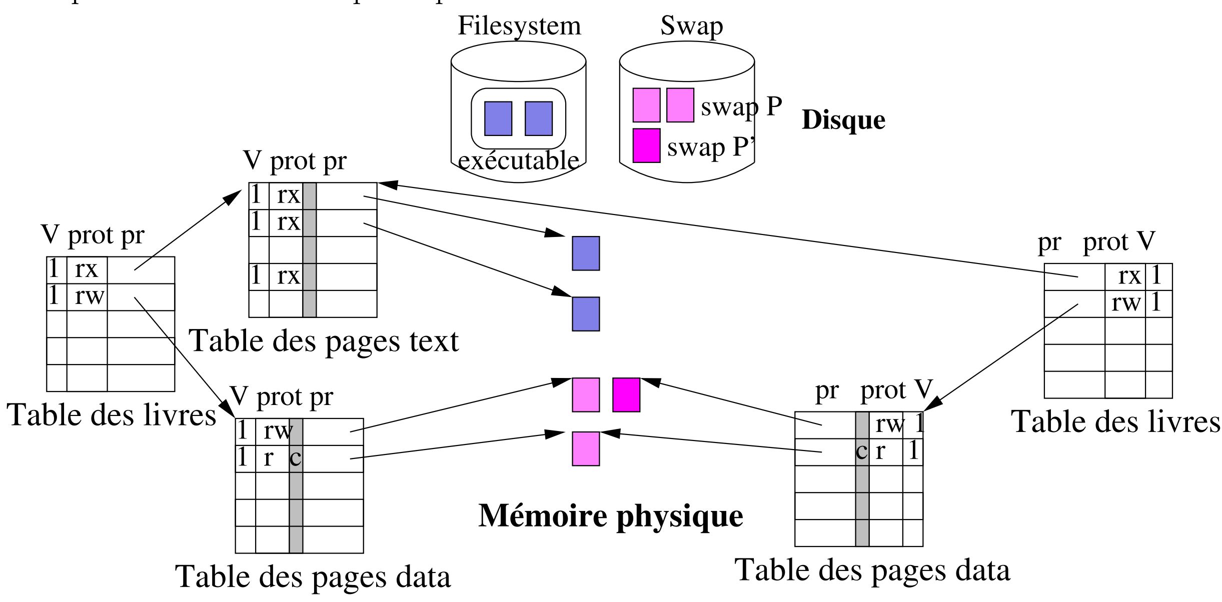
Beaucoup mieux : copy-on-write

- Partage des pages tant que non modifiées :
  - Protection contre l'écriture (bits de protection)
  - Marquage des pages comme étant "copy-on-write"
- Duplication paresseuse à la première écriture

Juste après le fork :



Après accès en écriture par le processus forké :



Autres utilisations du copy-on-write

- Recopie paresseuse de messages
- Mémorisation de points de reprise (checkpoint) incrémentale
- Ramasse-miettes copiants
- etc, ...

2 Edition de liens dynamique : un survol

Rappels sur la liaison

- Objets *logiques* (variables, procédures, fichiers)
- Objets *physiques* (valeurs, emplacements mémoire)
- *Liaison* : passage du
  - *nom de l'objet logique* (identificateur de variable, procédure)
  - à sa représentation concrète, au moins *les noms des objets physiques* supportant cette représentation.

Logiciels contribuant à la liaison (vus en licence) :

- *Traducteur* (compilateur ou assembleur).
  - Traduction de code source en code machine
  - Traduction des identificateurs d'objets dans une représentation interne
- *Editeur de liens* : regroupement du code et des données de plusieurs modules en résolvant les références externes
- *Chargeur* : initialisation de la machine (processeur + mémoire) pour que le programme puisse être exécuté

2.1 Edition de liens statique vs dynamique

(Instant ou les identificateurs sont associés à des adresses)

- A l'*écriture* du programme. Nom des objets physiques dans le texte source (généralement assembleur)
- A l'*édition de liens*. L'adresse d'implantation des programmes est fixée par l'éditeur de liens et pas par le chargeur
  - Systèmes à pagination : l'adresse d'implantation est une adresse *virtuelle*
  - Systèmes à adressage réel : l'adresse d'implantation est une adresse *réelle*
- Au *chargement* : l'adresse d'implantation est fixée au chargement pas ne change pas pendant l'exécution
- A l'*exécution* (liaison *dynamique*) : les adresses sont fixées à l'exécution, le programme ne contient plus aucune adresse et peut donc toujours être déplacé en cours d'exécution

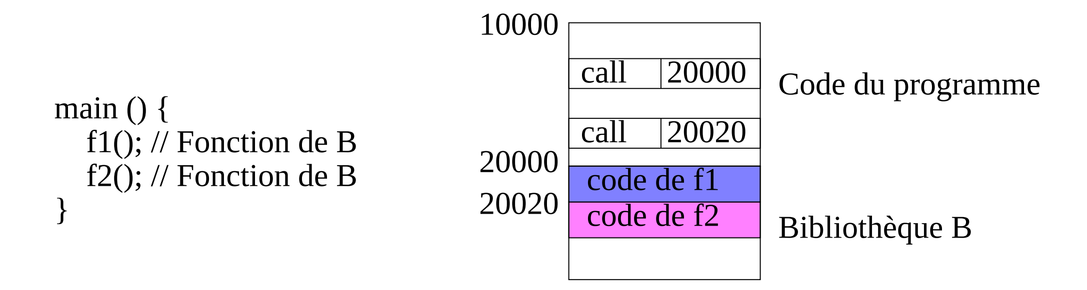
Moment de la liaison

Remarques 36.
- *Plus la liaison est tardive, meilleure sera l'adaptation du programme à une évolution de son environnement d'exécution*
- *Plus la liaison est tardive, plus les informations nécessaires à la liaison devront être conservées longtemps*

- Edition de liens *statique* : tous les identificateurs ont été traduits avant exécution (même si la liaison n'est pas tout à fait terminée)
- Edition de liens *dynamique* : il reste des identificateurs de références externes non résolus au début de l'exécution

Edition de liens statique

Tous les identificateurs ont été transformés en adresses *avant exécution*



Limites de l'édition de liens statique

- *Liaisons inutiles* quand des objets sont liés et non utilisés (appels conditionnels)
- *Gestion des évolutions* difficile (versions, corrections de bugs). Nécessité de refaire l'édition de liens pour bénéficier d'une nouvelle version
- *Consommation d'espace disque et mémoire* inutiles pour les copies des objets liés

2.2 Edition de liens dynamique

Edition de liens dynamique

Instants possibles de la liaison :

- *Au chargement* du module contenant une référence externe, ou
- A la *première référence* à un objet externe

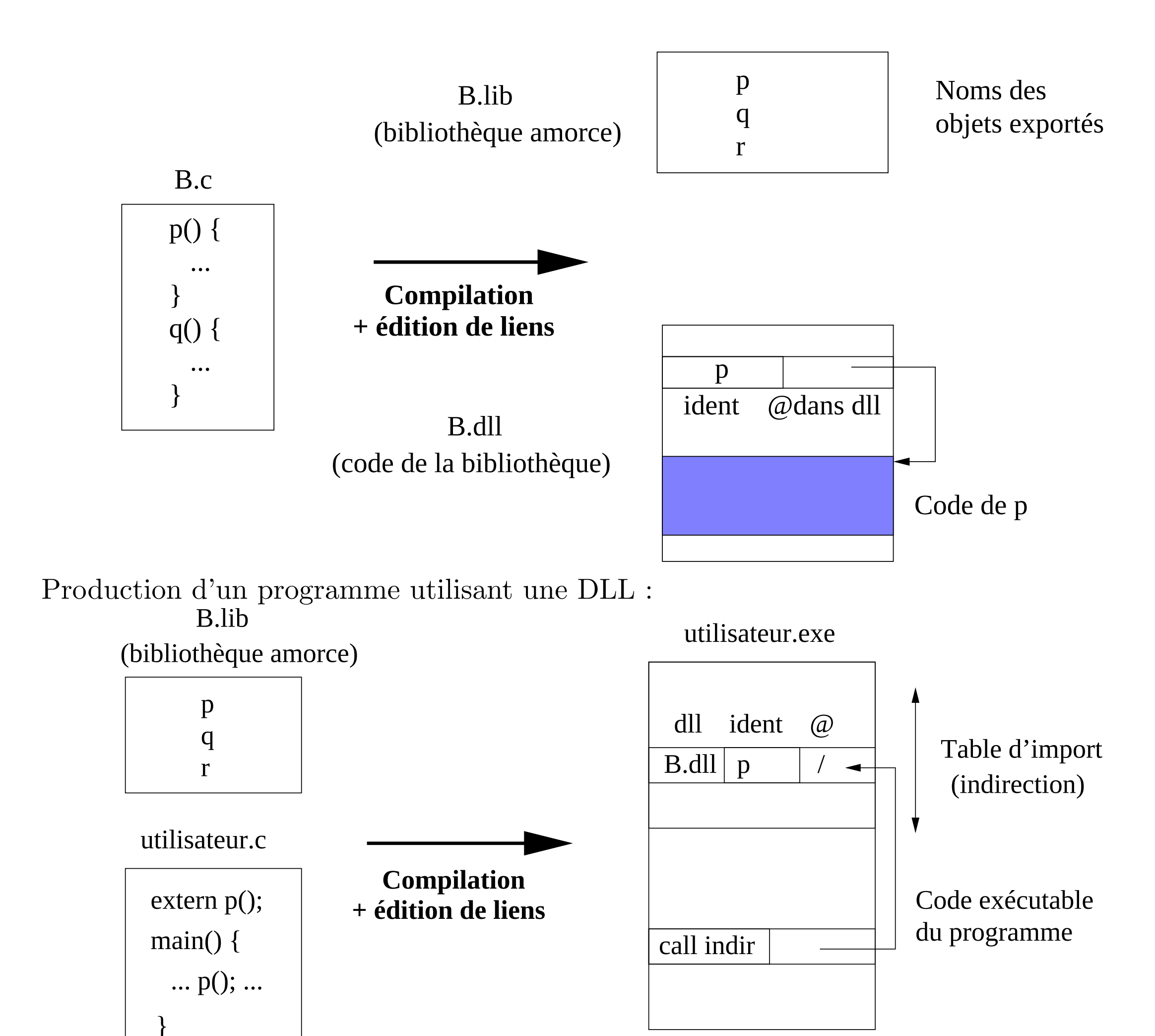
Remarque 37.

- *Edition de liens dynamique va de pair avec partage des bibliothèques $\Longrightarrow$ les bibliothèques partagées doivent être réentrantes*
- *Edition de liens dynamique $\Longrightarrow$ les symboles non résolus doivent être conservés plus longtemps qu'avec une édition de liens statique*
- *Résolution des liens inconnus plus tardive ...*

Principe :

- Edition de liens statique avec une *bibliothèque amorce* qui contient un élément par fonction non résolue
- Initialisation de cette table *au chargement* du programme utilisant la bibliothèque
- Exemple : DLL (Dynamic Link Library) des systèmes Windows

Production d'une DLL :



Production d'un programme utilisant une DLL :

⟹ *pas d'incorporation de la bibliothèque au programme exécutable*

Liaison (au chargement du programme) :

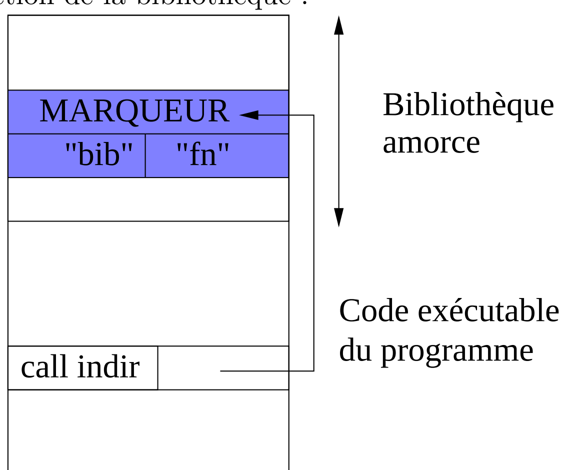
- Implantation de la bibliothèque dans l'espace d'adressage du processus (réservation table des pages)
- "Chargement" bibliothèque si c'est le premier processus à l'utiliser
  - Chargement partie résidente
  - Initialisation table des pages pour partie paginée
- Remplissage de la table d'importation avec les informations du fichier DLL

Remarques 38.
- *Liaison des bibliothèques même si elles ne sont pas utilisées pendant l'exécution*
- *Solution dans les systèmes Windows : liaison explicite des bibliothèques*
  - *pas de liaison statique avec la bibliothèque amorce, ni table d'indirection*
  - *fonctions système LoadLibrary, GetProcAddress, FreeLibrary*
  - *effort de programmation*
- *Fonctions appelées au chargement et déchargement des bibliothèques*
- *Fonctions similaires dans les systèmes UNIX (dlopen, dlclose, dlsym, dlerror)*

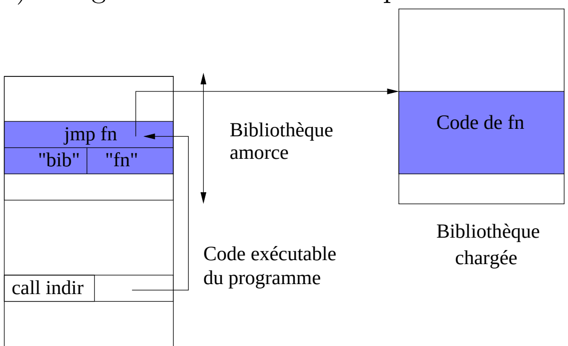
Principe :

— Edition de liens statique avec une *bibliothèque amorce* qui contient un élément par fonction non résolue
— Chaque référence externe provoque une exception pour *défaut de lien* (défaut de page, déroutement vers le superviseur)
— Résolution du défaut de liens dans la routine de traitement de cette exception :
  — Chargement de la bibliothèque si nécessaire
  — Remplacement du code déclenchant l'exception par un code d'appel

Avant appel d'une fonction de la bibliothèque :



Après appel et (éventuel) chargement de la bibliothèque :

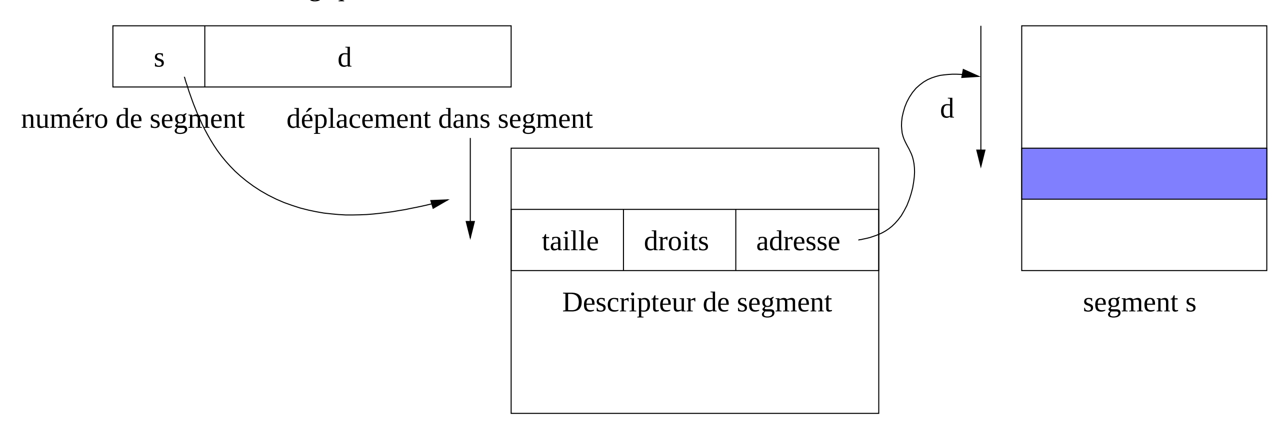


3 Espace virtuel segmenté

3.1 Segment

Segmentation

— Variation de l'adressage par registre de base
— *Segment* = unité de *structuration*, *partage* et *protection* de l'information
— Adresse segmentée = couple (nom_segment,déplacement)
— *Descripteur de segment* : contient les informations de taille, protection, et l'adresse d'implantation du segment
— Tentative d'accès invalide (droits, longueur) $\Longrightarrow$ déroutement vers le système d'exploitation



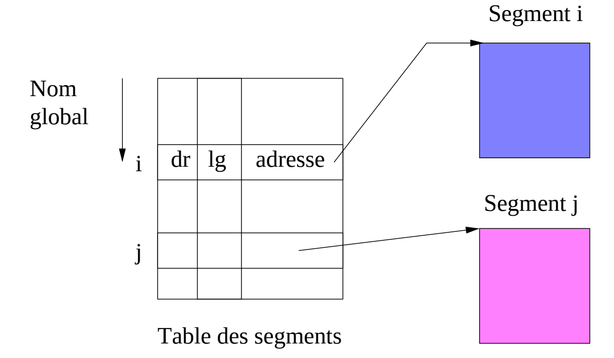
Remarque 39. — *Un segment constitue un espace d'adressage indépendant de l'espace d'adressage des autres segments*
— *Pas de rapport entre le dernier emplacement du segment i et le premier emplacement du segment $i+1$*

3.2 Organisation de la table des segments

Objectifs :
- Partager les segments $\Longrightarrow$ ne pas dupliquer l'information commune (taille, adresse d'implantation)
- Permettre d'exprimer des droits d'accès différents selon les processus

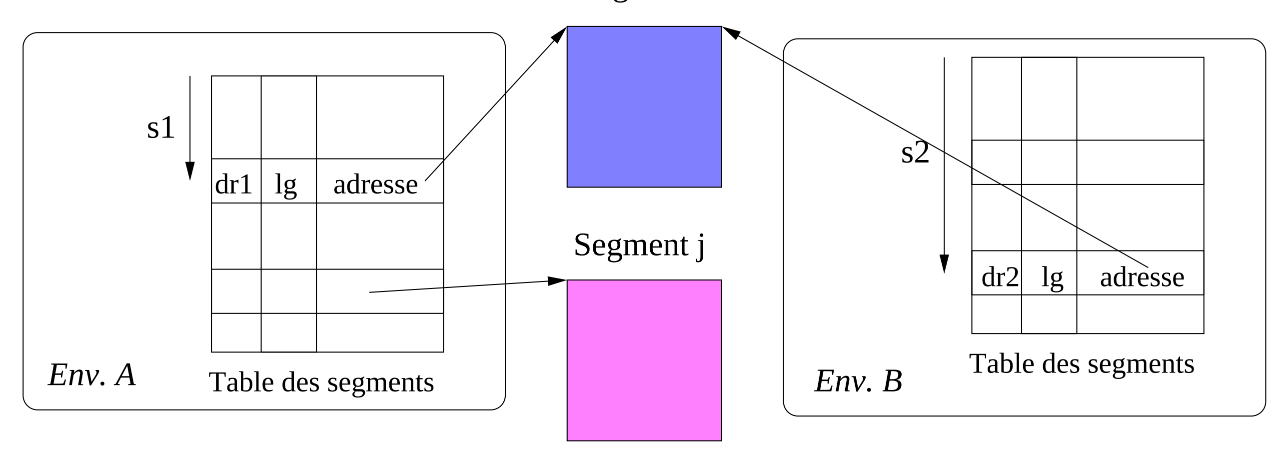
Organisations possibles :
- Table unique
- Tables multiples
- Organisation mixte

Organisation de la table des segments Table unique
- Environnement de désignation universel
- Nom de segment = *nom unique*, identique pour tous les utilisateurs
- Descripteur de segment unique $\Longrightarrow$ mêmes droits d'accès pour tous les utilisateurs
- Segment partagé $\Longrightarrow$ même adresse virtuelle dans les processus le partageant



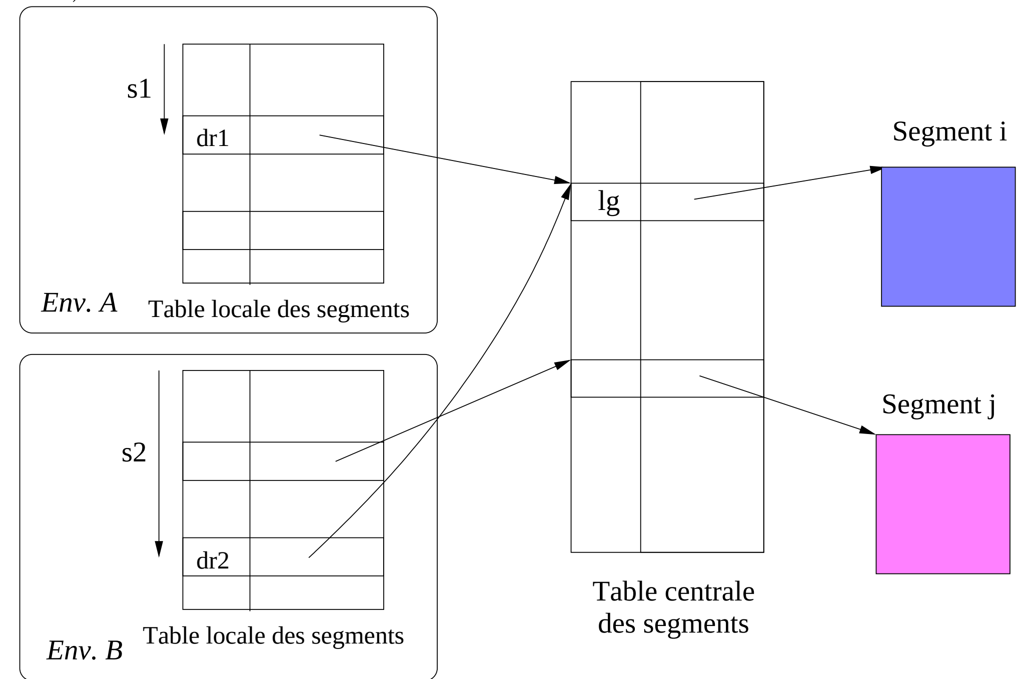
Organisation de la table des segments Tables multiples

- Plusieurs environnement de désignation (en général, un par processus)
- Nom de segment = *nom local* à l'environnement de désignation
- Segment partagé peut être vu à deux adresses virtuelles différentes



Organisation de la table des segments Organisations mixtes

- Informations indépendantes de l'environnement (longueur, adresse) dans un descripteur central unique
- Une table par environnement donnant les caractéristiques propres à l'environnement (droits d'accès)



Cinquième partie

Système de gestion de fichiers

1 Rappels sur les SGF

Système de gestion de fichiers (SGF)

- Gestion et accès à des informations stockées en dehors de la mémoire centrale
- Supports assurant la *persistance* de l'information

Fichier

Definition 40 (Fichier)**.**
- "Réservoir" d'informations stockées sur un support de stockage permanent
- Rôles :
  - *stockage permanent* : conservation d'informations sur une longue durée
  - *communication* : échange d'informations entre usagers ou entre programmes

Système de gestion de fichiers (SGF)

Interface :
- Création, destruction
- Ouverture ou fermeture (session de travail sur le fichier)
- Positionnement dans le fichier
- Lecture et écriture
- Gestion des répertoires et des droits d'accès

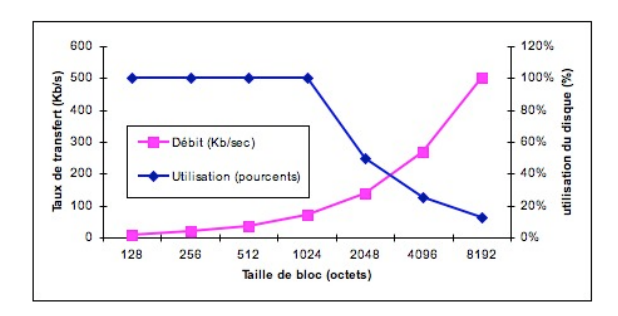
Interface système + bibliothèque des langages (ex : E/S tamponnées dans la bibliothèque standard C)

Problèmes à résoudre :
- Gestion de l'espace disque (création, allongement)
- Gestion des accès au contenu
- Gestion de l'ensemble des fichiers (nommage, hiérarchie de fichiers, contrôle d'accès)

1.1 Gestion de l'espace disque

- Unité d'allocation (bloc ou granule) : suite de secteurs consécutifs de la même piste. Bloc = plus petite unité allouable
- Crières de choix de la taille d'un bloc :
  - performances de l'allocation de bloc (temps d'allocation, taille de la structure représentant l'état du disque)
  - fragmentation interne
  - performance des accès disque

Exemple 41. Pistes de 128 Ko, rotation en 8 ms, positionnement bras en 10 ms, fichiers de la taille moyenne 1 Ko



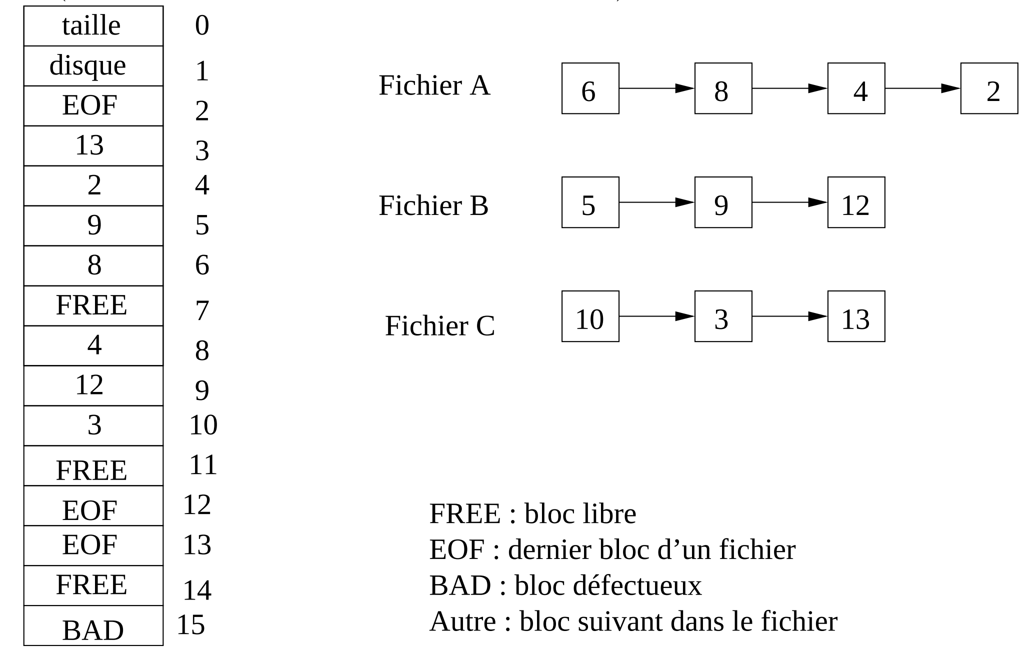
Allocation des fichiers sur disque :

- Allocation *contiguë* : blocs consécutifs de la même piste et/ou pistes adjacentes
  - diminution des nombres de mouvements du bras
  - structure d'implantation d'un fichier simple (@début + taille)
  - difficulté d'allocation de l'espace disque (idem gestion de mémoire par zones de taille quelconque)
- Allocation *non contiguë* : blocs disque répartis sans contrainte
  - allocation de l'espace disque simple (zones de taille fixe)
  - description de l'implantation d'un fichier sur disque est plus complexe
  - risque d'avoir plus de mouvements du bras
  - $\Longrightarrow$ défragmentation

Exemple 42 (La FAT – File Allocation Table – MS-DOS). — Table unique contenant : les blocs disque libres + les listes des blocs disque des fichiers

- Conçu à l'origine pour les disquettes de 320K, passe mal à l'échelle (ne peut plus être stockée intégralement en mémoire)

Exemple 43 (La FAT – File Allocation Table – MS-DOS).



1.2 Mise en œuvre des accès

- *Structuration logique* des informations (suite de caractères, de structures, etc) - en général, non structuré
- *Fonctions d'accès logiques* : accès direct, accès séquentiel, accès indexé
- *Entrées/sorties physiques* : par blocs de taille fixe
- Un des rôles du SGF : mettre en œuvre les E/S logiques en utilisant les E/S physiques

Mise en œuvre des accès Structures de données

- *Table d'implantation du fichier* : informations de mise en œuvre du fichier (taille, organisation logique)
  - Informations *permanentes*
  - Chargées en mémoire à l'ouverture pour accélérer l'accès au fichier
  - Exemple : inode Unix, FAT MS-DOS
- *Bloc de contrôle d'entrée/sortie* (descripteur, file handle) : informations liées par les accès en cours (prochain article à lire pour les fichiers à accès séquentiel, tampons d'entrée/sortie)
  - Durée de vie = durée d'ouverture d'un fichier
  - En général, une copie par processus ayant ouvert le fichier

Mise en œuvre des accès Cache disque

- Unité de transfert = bloc
- Géré entièrement par logiciel
- Intérêt
  - Tout accès logique n'entraîne pas d'accès physique
  - Permet les politiques de préchargement en cas d'accès séquentiel
- Politique de recopie
  - recopie *immédiate* : E/S physiques inutiles, mais disque toujours à jour
  - recopie *retardée* : moins d'E/S physiques, mais disque n'est pas toujours à jour ($\Longrightarrow$ problèmes en cas de défaillance, d'utilisation de fichiers pour communiquer)

  $\Longrightarrow$ compromis : recopies périodiques, fonctions de vidage (sync)
- Politique de remplacement : taille du cache disque + espacement des accès $\Longrightarrow$ on peut envisager une politique LRU

1.3 Désignation

Désignation

Propriétés des noms :

- *Durée de vie* : permanent ou temporaire
- *Portée* : globale ou locale
- *Nature de l'utilisation* : utilisateur ou système

Types de noms utilisés dans un SGF :

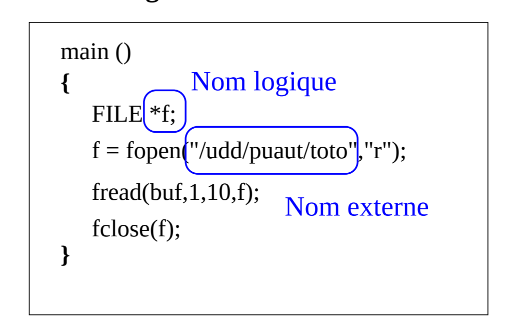
- *Nom externe* : nom donné par l'utilisateur (permanent, global, utilisateur)
- *Nom interne* : nom utilisé par le système pour désigner l'ensemble des informations du fichier (permanent, global, système)
- *Nom logique* : nom utilisé par l'utilisateur pour désigner le fichier ouvert (temporaire, local, utilisateur)

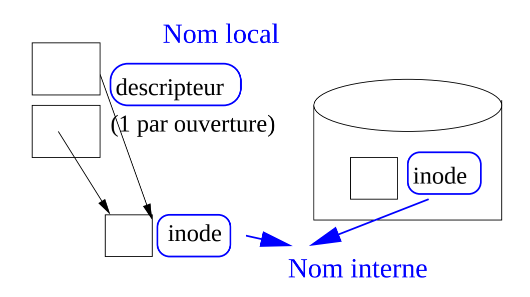
— *Nom local* : identification système du bloc de contrôle d'entrée/sortie (temporaire, local, système)

Types de noms utilisés dans un SGF :

	Nom global permanent	**Nom local temporaire**
Utilisateur	Nom externe	Nom logique
Système	Nom interne	Nom local

Exemple 44 (Exemple d'Unix). Noms et liaison des noms dans Unix





2 Le partage des fichiers

2.1 Contrôle des accès simultanés

Problème :

— Exécution parallèle (ou pseudo-parallèle) des processus

— Les processus peuvent accéder au(x) même(s) fichier(s)

$\Longrightarrow$ Cohérence du contenu des fichiers

$\Longrightarrow$ Synchroniser les accès aux fichiers

Exemple 45.

```
FILE *f = fopen("toto,"r");
for (int i=0;i<N;i++) {
    fread(buf,1,sizeof(int),f);
}
fclose(f);
```

```
FILE *f = fopen("toto,"w");
for (int i=0;i<N;i++) {
    fwrite(&i,1,sizeof(int),f);
}
fclose(f);
```

Classes de politiques de contrôle :

— Contrôle à l'*ouverture* vs contrôle lors des *accès élémentaires* (lecture/écriture)

— Politique de contrôle *systématique* (ex : lecteur/rédacteur) vs politique de contrôle laissée à l'utilisateur (verrous)

Exemples de politiques de contrôle :

— Politique systématique à l'ouverture : interdire deux ouvertures simultanées du même fichier

— Politique utilisateur au niveau des accès élémentaires en utilisant des verrous sur des portions du fichier (UNIX)

2.2 Protection

Propriétés à assurer :

— *Confidentialité* : empêcher la divulgation des informations sans autorisation

— *Intégrité* : empêcher la corruption des données par des fautes (accidentelles ou intentionnelles)

— *Disponibilité* : l'utilisateur peut accéder au service offert

— *Fiabilité* : le service rendu est correct

$\Longrightarrow$ Domaine général de la *sûreté de fonctionnement* (ici, on s'intéressera principalement à la confidentialité et à l'intégrité

Eléments nécessaires pour assurer la confidentialité et l'intégrité :

— *Mécanisme d'authentification* : moyens de s'assurer de l'identité d'un usager (ex : mot de passe)

— *Mécanisme de contrôle d'accès* : moyens de limiter les accès aux objets

— *Politique de sécurité* : règles sur la façon d'accorder des droits aux usagers

$\Longrightarrow$ Accent mis par la suite sur les mécanismes de contrôle d'accès

Domaines et droits d'accès

— *Sujet* : entité possédant des droits d'accès (processus, s'exécutant pour le compte d'un utilisateur au sens large)

— *Objet* : entité à protéger (fichier, zone de mémoire, etc)

— *Domaine de protection* : ensemble d'objets accessibles à un instant donné et droits d'accès associés, couples *(objets,droits)*

Matrice de droits : $droits[i, j]$=ensemble des droits sur l'objet j quand on est dans le domaine i.

Exemple 46.

	f1	**f2**	**f3**	**imprimante**
Domaine 1	lire, écrire	écrire		
Domaine 2		lire, écrire	lire, écrire	
Domaine 3	lire	écrire		utiliser

Représentation des domaines

La matrice des droits est *grosse* et *vide* $\Longrightarrow$ il faut trouver un moyen de stocker uniquement les cases "pleines" :

— Par colonne (par objet) : mécanisme de *liste de contrôle d'accès* (liste de couples *(domaine, droits)*)

— Par ligne (par domaine) : mécanisme de *capacité* (liste d'objets accessibles par domaines)

3 Exemple : le SGF d'UNIX

Fichiers UNIX

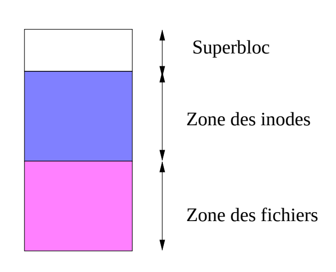
- Suite d'*octets*
- Appels systèmes : lecture/écriture de séquences d'octets, positionnement dans le fichier
- Structure de fichiers *très simple*. Définition de structures plus complexes (notion d'article, accès indexé, ...) laissée au niveau utilisateur (directement, bibliothèque)
- Interface fichier utilisée pour tout objet ayant un nom externe visible dans la hiérarchie des fichiers (périphériques, tubes, etc)

Volume

- Représentation bas niveau d'un disque ou d'une partie de disque
- Organisé logiquement comme une suite de blocs de taille fixe
- Constitue le support d'un système de gestion de fichiers

Notion de superbloc :

- Bloc spécial décrivant un volume
- Contenu d'un superbloc :
  - Taille du système de fichiers
  - Nombre de blocs libres
  - Liste des blocs libres
  - Nombre d'inodes libres
  - Liste des inodes libres

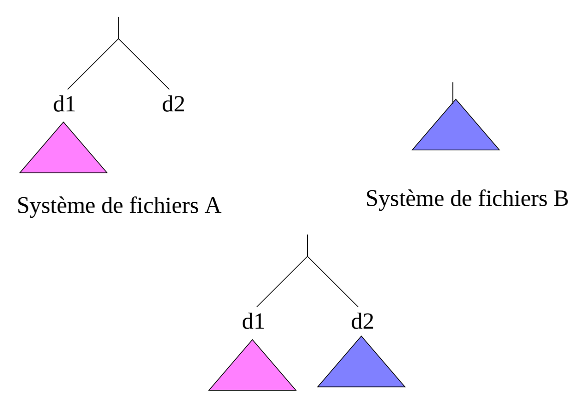


Structure d'un volume UNIX

Montage

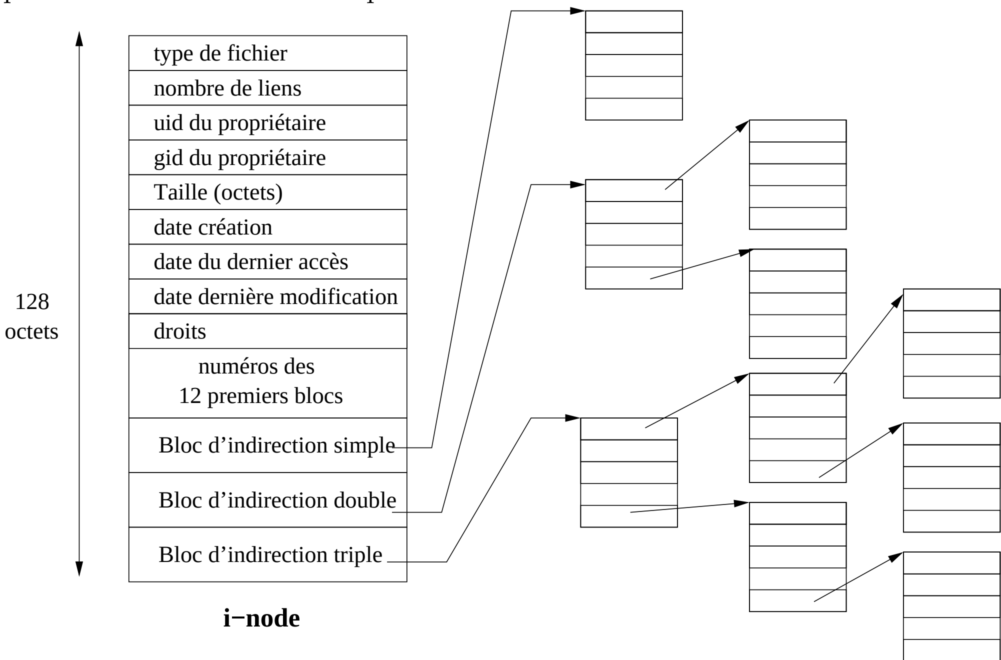
- Opération permettant d'intégrer plusieurs systèmes de fichiers dans une seule hiérarchie de désignation
- Commande *mount*

$\Longrightarrow$ Possible que deux utilisateurs n'aient pas la même hiérarchie de fichiers

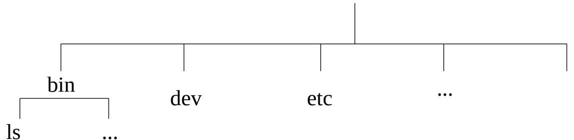


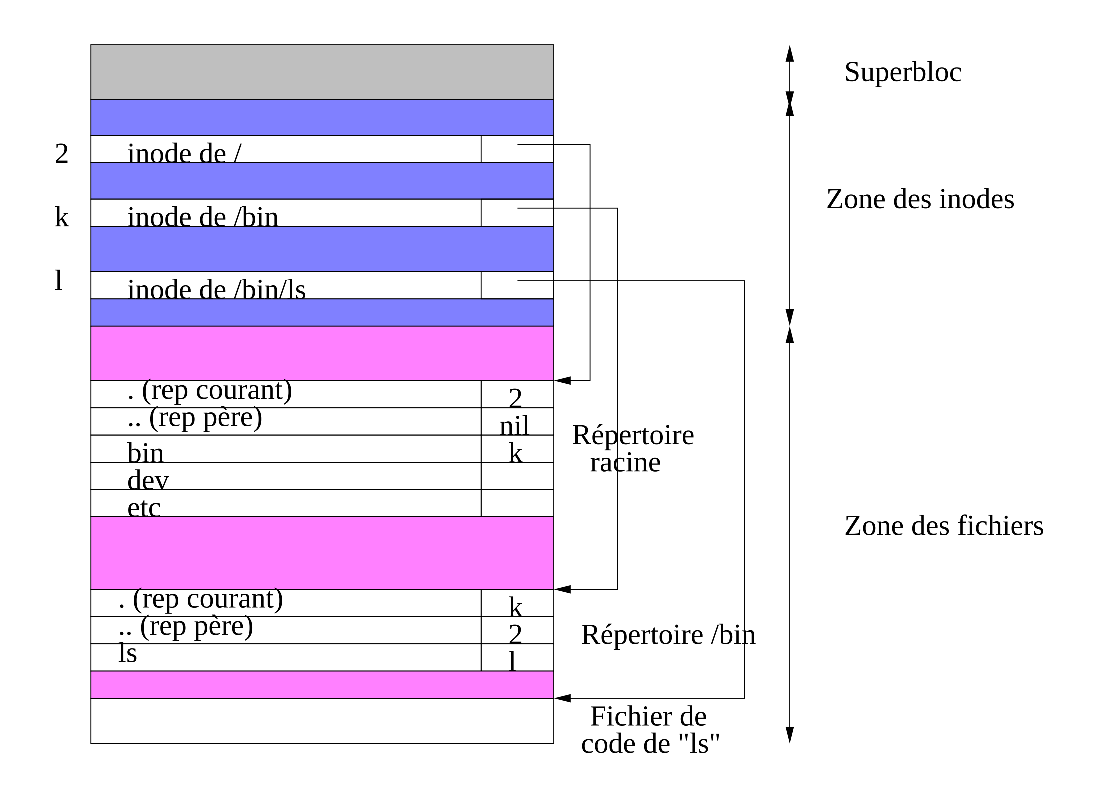
Représentation permanente des fichiers

- Inode
  - Informations de propriété et droits d'accès
  - Taille
  - Table d'implantation
  - Nom interne d'un fichier = couple *(n^o volume, n^o inode)*
- Répertoire : mémorise la correspondance entre nom externe et inode



Contenu du volume contenant le système de gestion de fichiers :





Représentation des fichiers utilisés (ouverts)

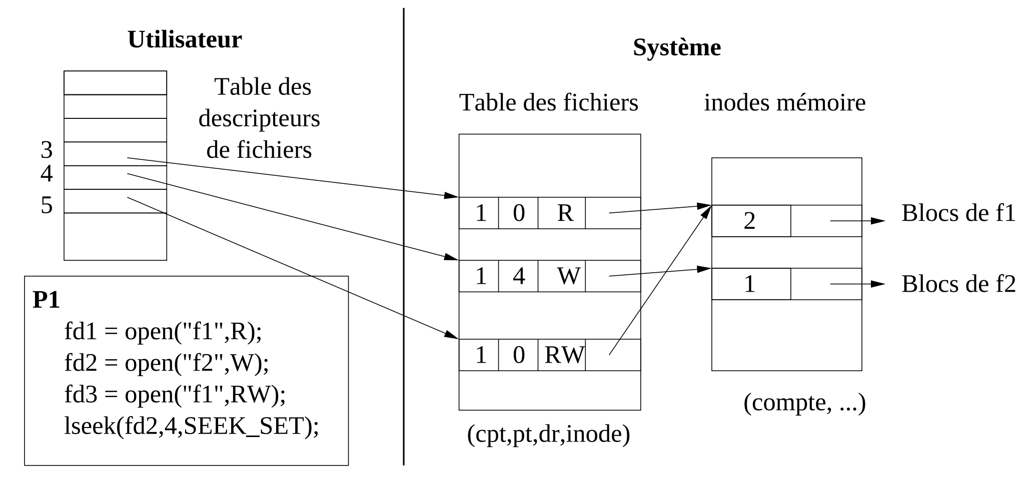
- *Inode mémoire*, global à tous les processus
  - Copie de l'inode disque
  - Compteur d'utilisation
  - Verrou (accès exclusif à l'inode)
- *Table des fichiers*, table *système* globale à tous les processus, contient pour chaque fichier ouvert :
  - Pointeur sur inode mémoire
  - Compteur d'utilisation
  - Droits d'accès pour cette ouverture
  - Pointeur de fichier (caractère courant)
- *Table des descripteurs de fichiers*, table *utilisateur* propre à un processus. Pour chaque fichier ouvert, pointe sur une entrée de la table des fichiers. Indices 0, 1 et 2 réservés (stdin, stdout, stderr)

Ouverture d'un fichier

fonction open (**chaine** nomfich, **acces** m) **resultat n^o descripteur**
debut
  Utilise *nomfich* pour retrouver l'inode disque et le copie en mémoire
  si fichier inexistant **ou** accès demandé interdit
  alors résultat erreur
  sinon
    alloue une entrée dans la table des fichiers et l'initialise
    alloue un descripteur de fichier et l'initialise

résultat numéro du descripteur alloué
fsi
fin

Exemple 47. Structures de données lors de l'ouverture d'un fichier



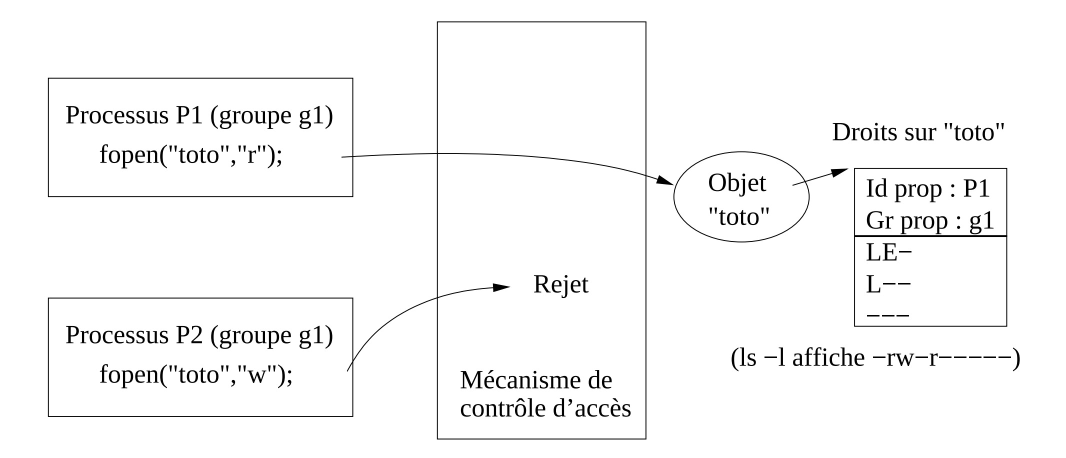
Remarques 48.
— fd1, fd2 *et* fd3 *sont des noms logiques, ils n'existent pas en dehors du programme. Il sont locaux au processus.*
— *Ces structures de données permettent de partager les inodes mémoire entre processus*

Contrôle des accès concurrents

— Initialement, aucun contrôle d'accès concurrents prévu
— Ajouts ultérieurs via l'appel système *fcntl*
— *Verrous* :
  — Possibles sur des *parties* de fichiers (offset+taille)
  — Verrous exclusifs ou partagés
  — Politique de contrôle à la charge de l'utilisateur ($\Longrightarrow$ possibilités d'interblocages)

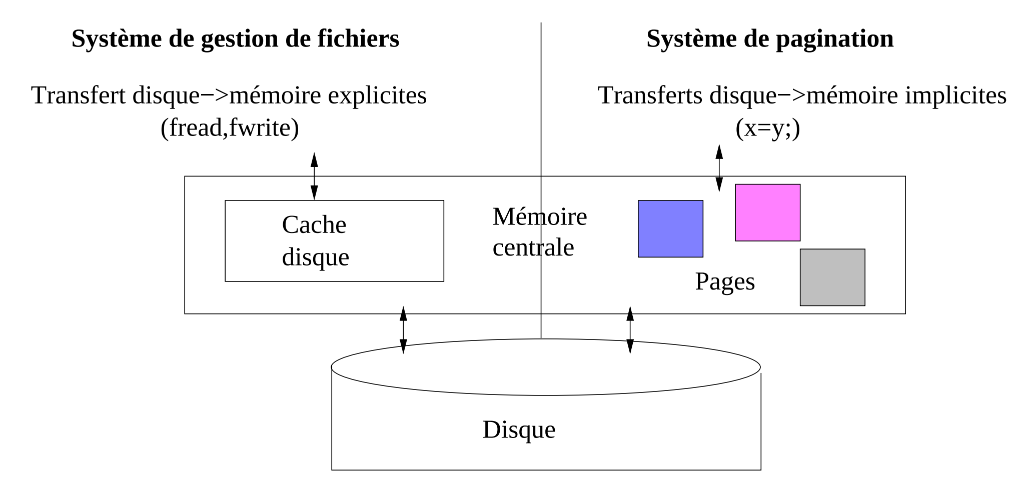
Protection

— Représentation simplifiée des droits (*"liste"* de taille fixée)
— Trois domaines d'utilisation possibles pour un fichier : *propriétaire*, *groupe*, *autres*
— Trois droits possibles pour chaque fichier normal : *lire*, *écrire*, *exécuter*
$\Longrightarrow$ 9 bits suffisent pour décrire tous les droits associés au fichier



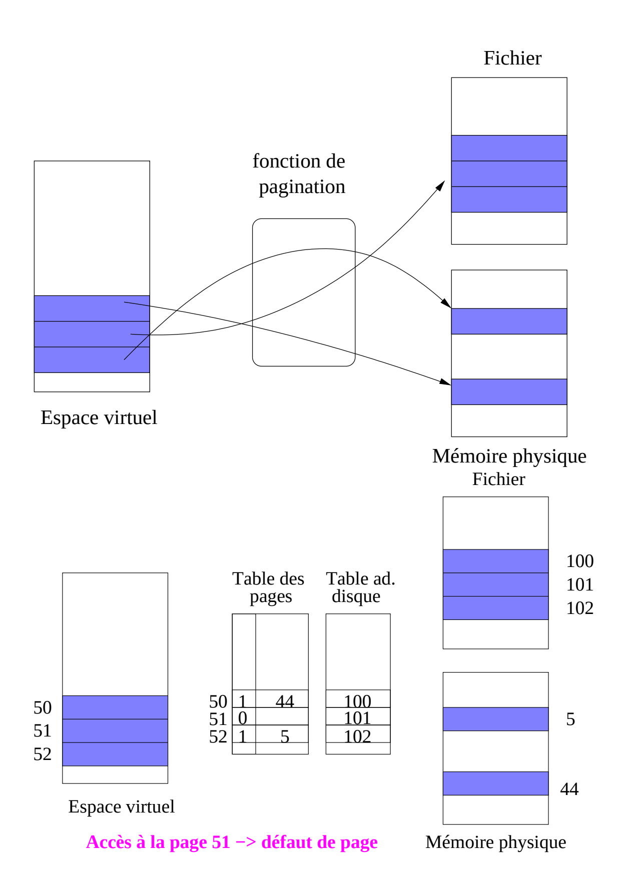
Remarque 49. *Pas possible de dire facilement que dans un groupe* $g1$*, seul* $u1$ *a certain droits (*$\Longrightarrow$ *comment faire ?)*

4 Pagination et gestion de fichiers



Fichiers mappés

- Taille de bloc disque = taille d'une page virtuelle
- Primitive (*mmap* UNIX) établissant une correspondance entre :
  - Une zone de mémoire virtuelle
  - Un fichier
- Utilisation de la zone de mémoire virtuelle comme une zone standard
- C'est l'algorithme de remplacement de page qui met en œuvre le cache disque



Fichiers mappés Interface UNIX

*void *mmap(void *ad, size_t l, int prot, int fl, int fd, off_t of) ;*

- ad : adresse de visibilité (0 $\Longrightarrow$ le système choisit)
- l : longueur zone à rendre visible, of = offset dans le fichier
- prot : droits d'accès
- fl : indique si en cas de modification le fichier lui-même est modifié (MAP_SHARED) ou si les modifications sont privées au processus (MAP_PRIVATE)
- fd : descripteur du fichier (ouvert)

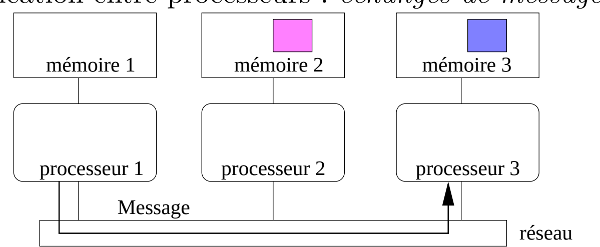
Exemple 50.

```
f = open("toto",O_RDWRITE) ;
char *ad = mmap(0,1024,PROT_WRITE,MAP_SHARED,f,0) ;
for (i=0 ;i<1014,i++) ad[i] = i ;
```

Sixième partie
Gestion de l'information dans les systèmes répartis

Architecture des systèmes répartis

— Machines connectées par un réseau
— Pas de mémoire commune
— Pas d'horloge commune
— Moyen de communication entre processeurs : *échanges de messages*

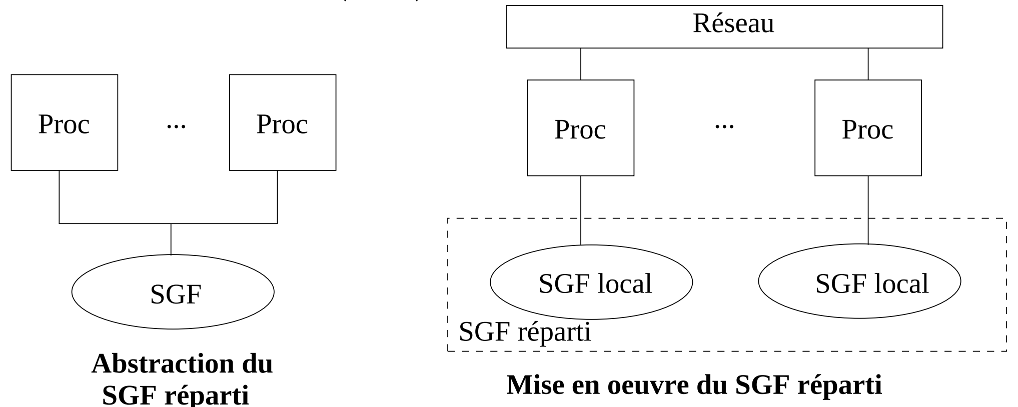


Gestion mémoire dans les systèmes répartis

Quelle machine virtuelle offrir à l'utilisateur ?

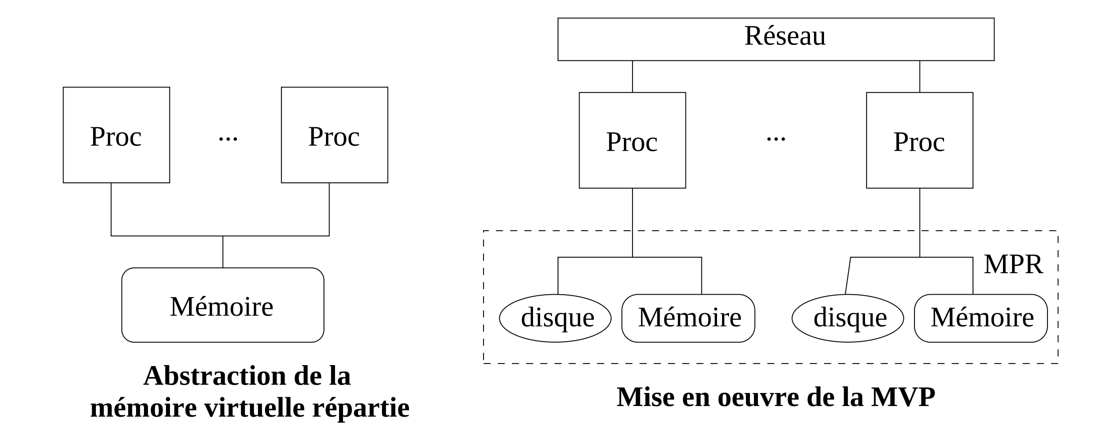
— Envoi de messages
  — Outils différents des outils habituels
  — Manipulation différentes des informations locales et distantes (sockets, RPC, RMI, etc.)
— Cacher les envois de messages : manipulation de données classiques, identiques pour les informations locales et distantes :
  — Fichiers : systèmes de gestion de fichiers répartis
  — Mémoire : mémoire virtuelle répartie

Système de gestion de fichiers (SGF) réparti



Abstraction du SGF réparti — **Mise en oeuvre du SGF réparti**

Mémoire virtuelle répartie



Abstraction de la mémoire virtuelle répartie

Mise en oeuvre de la MVP

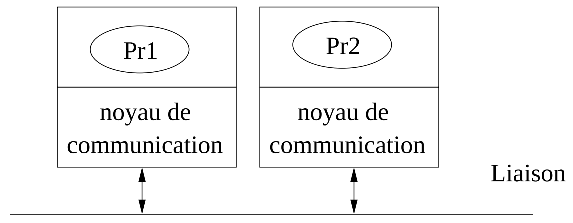
1 Systèmes de gestion de fichiers répartis

Accès à l'information distante

— *Non transparent* : utilisation de commandes réalisant une copie locale avant utilisation (ftp, rcp, etc.)

```
ftp nom-machine
    get nom-fich-distant nom-fich-local
    quit
<accès au fichier local grâce au SGF local>
```

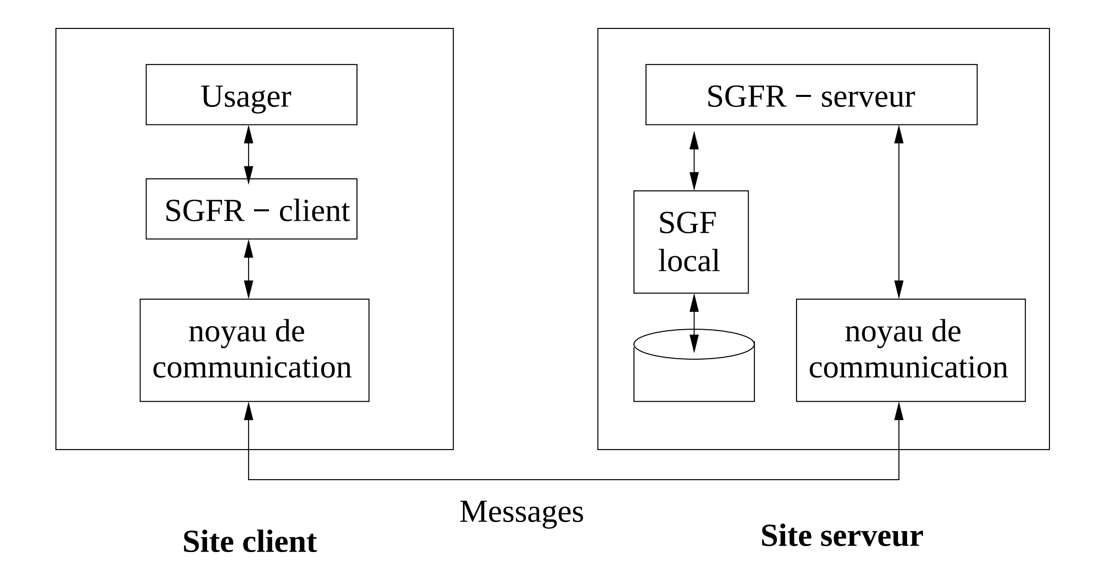
— *Transparent* : mécanismes semblables aux accès fichiers locaux : système de gestion de fichier réparti (ex : NFS)



Principe d'un SGF réparti

— Stations peuvent être spécialisées dans le stockage des fichiers (*serveurs de fichier*)

— Fonction de système de gestion de fichiers assure la *transparence d'accès* en engageant un dialogue avec le serveur de fichiers (échanges de messages via un noyau de communications)



1.1 Propriétés d'un SGF réparti

Propriétés d'un SGF réparti

Transparence à la distribution

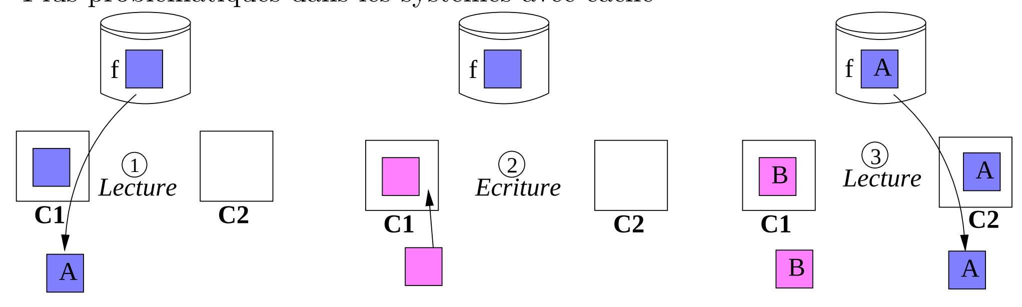
- *Transparence d'accès* : toutes les opérations applicables aux fichiers locaux sont applicables aux fichiers distants
- *Transparence à la localisation* : les utilisateurs voient un espace de noms uniforme ; les fichiers peuvent être déplacés sans changer de noms externes

Sûreté de fonctionnement

- *Fiabilité* : aucune donnée ne sera perdue ou corrompue suite à la défaillance d'un serveur
- *Disponibilité* : le service de fichiers sera toujours disponible en dépit de la défaillance d'un serveur
- *Transparence aux défaillances* : la défaillance d'un serveur sera transparente à ses clients

Gestion des accès concurrents

- *Cohérence stricte* : lecture retourne la dernière écriture
  - Facile à assurer pour les systèmes sans caches
  - Plus problématiques dans les systèmes avec cache



- *Cohérence de session* : copie locale du fichier à l'ouverture, modifications visibles aux autres uniquement à la fermeture
- *Cohérence faible* : une opération de lecture retournera une valeur ayant été écrite au préalable au même emplacement, sans savoir laquelle

1.2 SGF réparti : éléments de mise en oeuvre

Problèmes liés à la répartition d'un SGF

- *Localisation* : déterminer sur quelle machine se trouve un fichier
- *Constitution de l'espace des noms externes* : quelle est la structure de l'espace des noms vue par un utilisateur ? par quel mécanismes est-il construit ?
- *Mode d'accès aux informations* : comment est réalisée une lecture, une écriture ? comment minimiser les transferts ? comment gérer les caches ?
- *Disponibilité* : comment faire face aux fautes matérielles ? quel est l'impact de la duplication sur l'accès aux fichiers ? comment gérer les répliques ?
- *Sécurité* : comment assurer la confidentialité et l'intégrité en présence de machines sur lesquelles on peut "manipuler" le matériel et le logiciel ? comment authentifier un usager ?

Choix généraux de conception

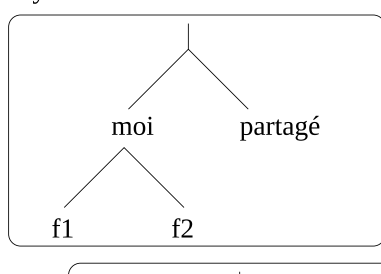
- Stations *banalisées* : toute station peut a priori être serveur de fichier ou client
- Stations *spécialisées* : une station ne peut pas être client et serveur de fichiers à la fois
  - permet de définir des règles de sécurité adaptées
  - exemple : serveurs de fichiers dans des locaux sûrs, confiance dans le matériel et le logiciel installés
- Serveurs *avec état* : le serveur stocke des informations sur les clients en cours d'utilisation des fichiers
  - Moins d'informations à transiter dans les requêtes
  - Facilité de lectures avec anticipation
  - Possible de gérer des verrous d'accès au fichier
  - Défaillance du client et du serveur peuvent laisser le système dans un état malsain (sessions jamais fermées, sessions fermées de manière autoritaire)
- Serveurs *sans état* : le serveur ne mémorise rien sur ses clients
  - Exemple : NFS

Espace des noms externes

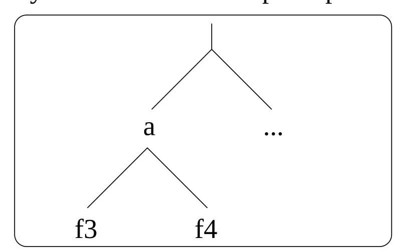
A. Inclusion du nom du serveur dans les noms externes (*/serveur/usr/fich*)
- Localisation du fichier triviale
- Migration difficile : pas de transparence à la localisation

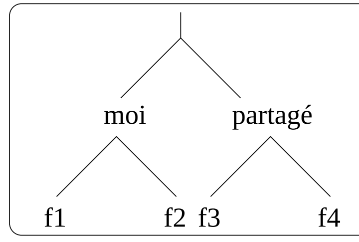
B. Montage à distance

Système de fichiers local de A



Système de fichiers exporté par B





Hiérarchie après montage à distance de a sous le le répertoire partagé de A

— Liaison (nom externe,localisation) dynamique, re-calculée à chaque montage. Localisation relativement simple
— Migration possible avec modification des séquences de montage
C. Espace de nommage unique indépendant de la localisation :
— Localisation recalculée dynamiquement (au moins à chaque ouverture)
— Migration dynamique possible
$\Longrightarrow$ Localisation plus complexe

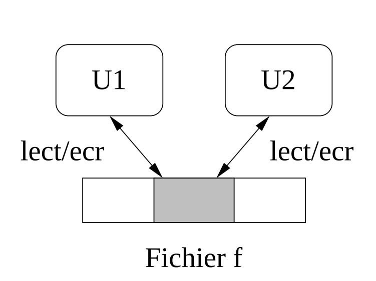
Localisation

— Noms internes doivent permettre de désigner le fichier dans *tout le système distribué* $\Longrightarrow$ *identificateurs uniques* (UID - Unique Identifiers)
— Le nom unique peut contenir une aide à la localisation (exemple : site de création)

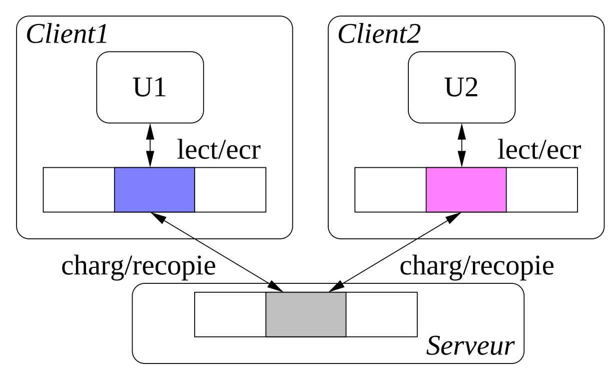
Gestion des caches

Envoi systématique d'une requête au serveur *inefficace* (latence réseau + disque) $\Longrightarrow$ utilisation de caches

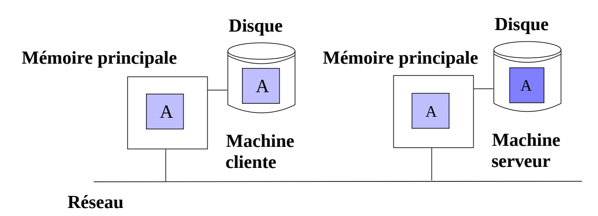
Abstraction



Mise en oeuvre



Localisation possible des caches :



- Granularité du cache
  - Bloc
  - Plusieurs blocs
  - Fichier complet
- Taille plus importante
  - taux de présence dans le cache élevé
  - temps de chargement important
  - problème de cohérence de cache

Politique de recopie

- *Ecriture immédiate*
  - Coûteux (latence, bande passante réseau)
  - Sémantique claire en présence de défaillance
- *Ecriture retardée*
  - Bonnes performances
  - Quand les fichiers sont détruits rapidement, le serveur n'en a pas connaissance
  - N'autorise qu'une cohérence faible
  - Sémantique peu définie en présence de défaillance
- *Ecriture à la fermeture* : mise à jour du serveur uniquement à la fermeture du fichier
  - Bonnes performances
  - Permet de mettre en œuvre la cohérence de session
  - Optimisations possibles pour les fichiers de faible durée de vie
- Compromis : *recopies périodiques*

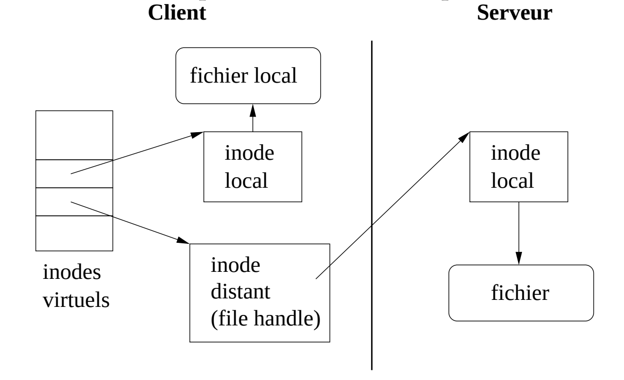
Cohérence des caches

- *Cohérence faible*
  - Aucun protocole nécessaire
  - Recopie régulière du cache client peut assurer que les valeurs lues ne seront pas trop anciennes
- *Cohérence de session*
  - Aucun protocole nécessaire
  - Seul problème : avoir assez de cache client pour conserver les modifications sur le site client ($\Longrightarrow$ cache client *sur disque* est le mieux adapté)
- *Cohérence stricte*
  - Nécessite un protocole particulier
  - Sprite : invalidation des caches client quand il existe au moins un écrivain sur un fichier (détecté à l'ouverture)
  - Echo : protocole à invalidation (n jetons de lecture ou *un* jeton unique en écriture)

1.3 Exemples

Network File System (NFS) Généralités

— Conçu par SUN en 1985, version 2 et 3 en 1989 et 1994
— Serveur sans état $\Longrightarrow$ le serveur ne gère pas les accès concurrents
— Montage à distance $\Longrightarrow$ pas nécessairement d'espace de nommage unique
— Communication client/serveur par appel de procédure à distance (RPC : Remote Procedure Call, équivalent RMI = Remote Method Invocation)
— NFS définit le protocole de communication client/serveur, pas l'implantation des clients/serveur
$\Longrightarrow$ haute disponibilité dépendante de l'implantation
$\Longrightarrow$ la gestion des caches n'est pas définie dans le protocole



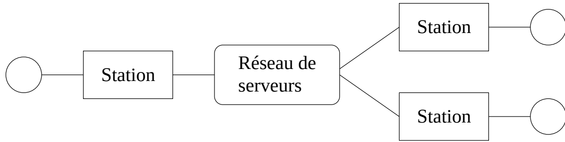
Network File System (NFS) Mise en œuvre SUN

Mise en œuvre des accès :
— Majorité des requêtes *idempotentes* (même résultat si exécutées plusieurs fois) $\Longrightarrow$ envoi répété des requêtes au serveur s'il ne répond pas (*NFS server not responding still trying*)
— Transferts de données par blocs *de grande taille* (8Ko)
— *Lecture avec anticipation* pour optimiser les accès séquentiels
— Fichiers lus intégralement si leur taille est inférieure à un certain seuil

Gestion des caches
— Caches clients *en mémoire* (un pour les répertoire, un pour les fichiers)
— Caches organisés *en blocs*, estampillés avec une date de dernière modification
— *Recopie retardée* des caches (périodiquement, toutes les 30 s)
— A l'ouverture d'un fichier contenu en cache, vérification de la validité de la copie
— *Invalidation périodique* des blocs (30 s). Recopie à la fermeture et flush.
$\Longrightarrow$ Pas de sémantique de cohérence précise

Andrew File System (AFS) Généralités

— Conçu par l'université de Carnegie Mellon dans le début des années 90 (1000 serveurs en 1996, 20000 clients répartis dans 10 pays)
— Ensemble de stations de travail avec disque connectées à un réseau de serveurs de fichiers



Andrew File System (AFS) Nommage et localisation

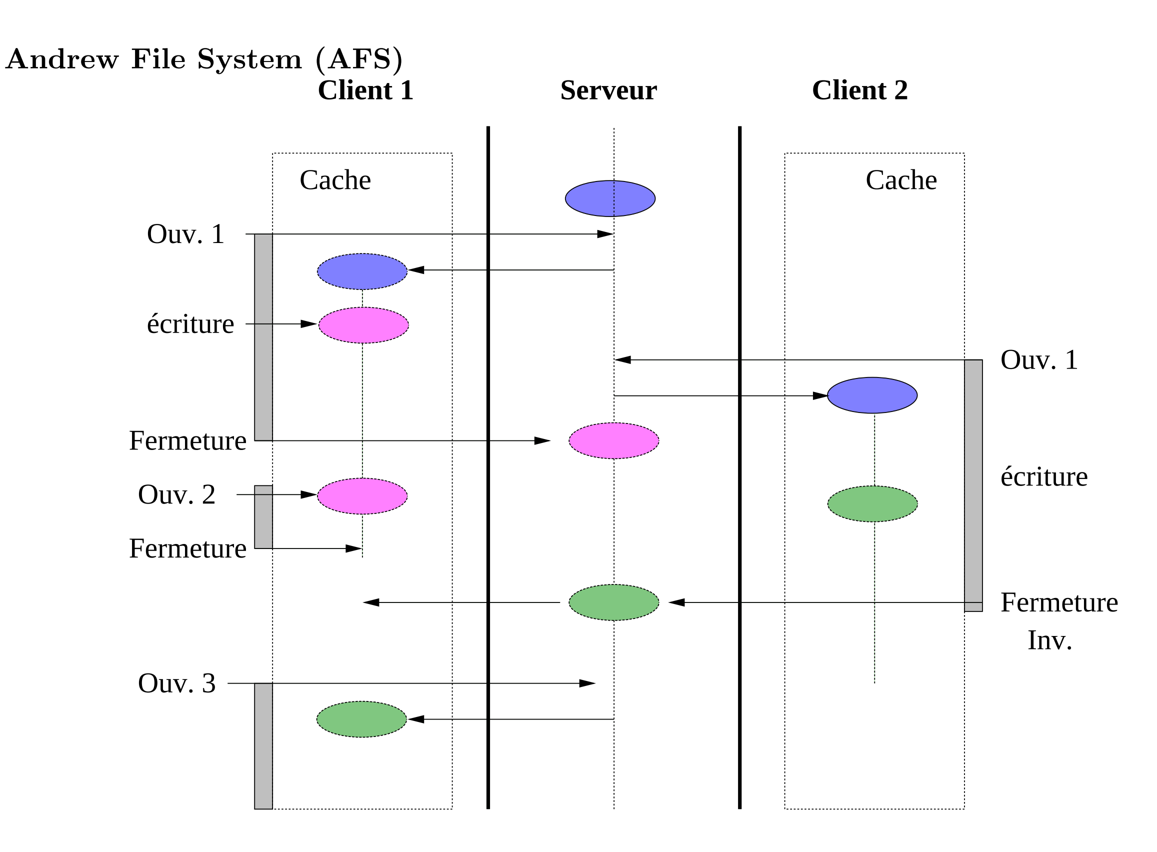
— Hiérarchie de fichiers *unique* intégrant tous les fichiers accessibles

— Noms internes uniques (*fid*)
— *Localisation dynamique.* Base de données dupliquée utilisée pour localiser les fichiers
— Protocole pour assurer la cohérence des copies de cette base

Andrew File System (AFS) Gestion des accès et caches

— Serveurs *avec état*
— *Disque local* des clients utilisé comme cache
— Sémantique d'accès de type *session*
  — Transfert du fichier complet en cache à l'ouverture
  — Accès entièrement locaux après ouverture
  — Recopie des données vers le serveur à la fermeture
  — A la fermeture, le client conserve une copie
  — Le serveur mémorise les clients ayant une copie. Demande d'invalidation envoyée quand le serveur reçoit une nouvelle version du fichier (fermeture par un autre client)

Andrew File System (AFS)

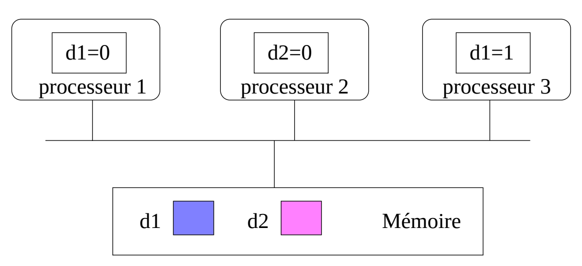


2 Mémoires virtuelles réparties

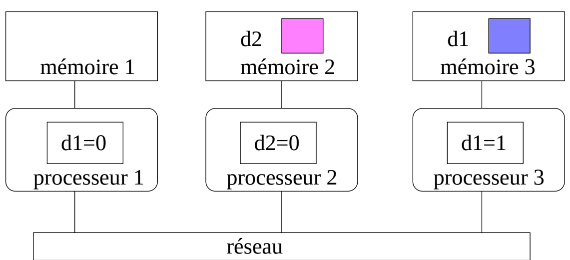
2.1 Principe

Principe des mémoires virtuelles réparties

Manipulation des données comme si elles étaient dans une mémoire unique partagée par tous les processeurs



En réalité, la mémoire partagée est mise en œuvre en utilisant les mémoires des processeurs constituant le système



2.2 Modèle de cohérence

Cohérence dans les mémoires virtuelles réparties

- *Modèle de cohérence*
  - Définit la/les valeur(s) pouvant être retournées par une opération de lecture (pas d'horloge commune)
  - Garanties fournies au programmeur
  - Garanties fortes $\Longrightarrow$ latence des accès mémoire élevée
  - Garanties faibles $\Longrightarrow$ latence plus faible
- *Protocole de cohérence*
  - Implantation particulière d'un modèle de cohérence
  - Un modèle de cohérence peut être implanté par plusieurs protocoles de cohérence

Modèles de cohérence Cohérences fortes

- *Cohérence stricte* : toute opération de lecture d'une variable partagée retourne la *dernière* valeur écrite dans cette variable
  - Demande l'existence d'un *ordre total* entre événements pour que le sens de *dernière* soit bien défini
  - Le maintien de cet ordre total est très coûteux
- *Cohérence séquentielle* : le résultat de toute exécution est le même
  - que si les opérations de tous les processeurs étaient exécutées dans un *ordre séquentiel* donné, et
  - que les opérations de chaque processus apparaissent dans cette exécution dans l'*ordre du programme*
  - Reformulation : tous les accès à la mémoire partagée seront vus dans le même ordre par tous les processus

Modèles de cohérence Cohérences faibles

- *Cohérence faible* : distinction des accès *de synchronisation* et des accès *ordinaires* :

  - les accès aux variables de synchronisation sont séquentiellement cohérents
  - accès à une variable de synchronisation n'est permis que quand tous les accès en écriture sur tous les processeurs sont terminés
  - accès à une variable ordinaire (lecture ou écriture) n'est permis que quand tous les accès aux variables de synchronisation sur tous les processeurs sont terminés
- *Cohérence à la libération* : distinction de l'acquisition des verrous de leur libération
  - Aquisition : attente de la propagation des modifications
  - Relâchement : propagation des modifications locales aux autres machines

2.3 Eléments de mise en oeuvre

Unité de transfert

- *Variable* (ou objet)
  - nécessité d'un support langage pour détecter l'absence d'une variable de la mémoire
- *Page* :
  - l'absence d'une page peut être détectée par le mécanisme de *défaut de page*
  - risque de *faux partage* : données non reliées peuvent être allouée dans la même page $\Longrightarrow$ problème de performance (effet ping-pong)

Réplication et migration

- *Duplication* : duplication de la donnée pour autoriser les manipulations parallèles sur des processeurs différents
- *Migration* : la donnée migre dans la mémoire du site demandeur. Mémoire = cache de la mémoire virtuelle répartie
- Classification des MVP :
  - *SRSW (Single Reader, Single Writer)* : pas de duplication. Efficacité limitée car aucune parallélisation des accès concurrents sur des nœuds différents
  - *MRSW (Multiple Reader, Single Writer)* : accès concurrents en lecture autorisés, mais pas en écriture
  - *MRMW (Multiple Reader, Multiple Writer)* accès concurrents en lecture et écriture autorisés

Gestion de la localisation et des accès

Structures de données nécessaires :

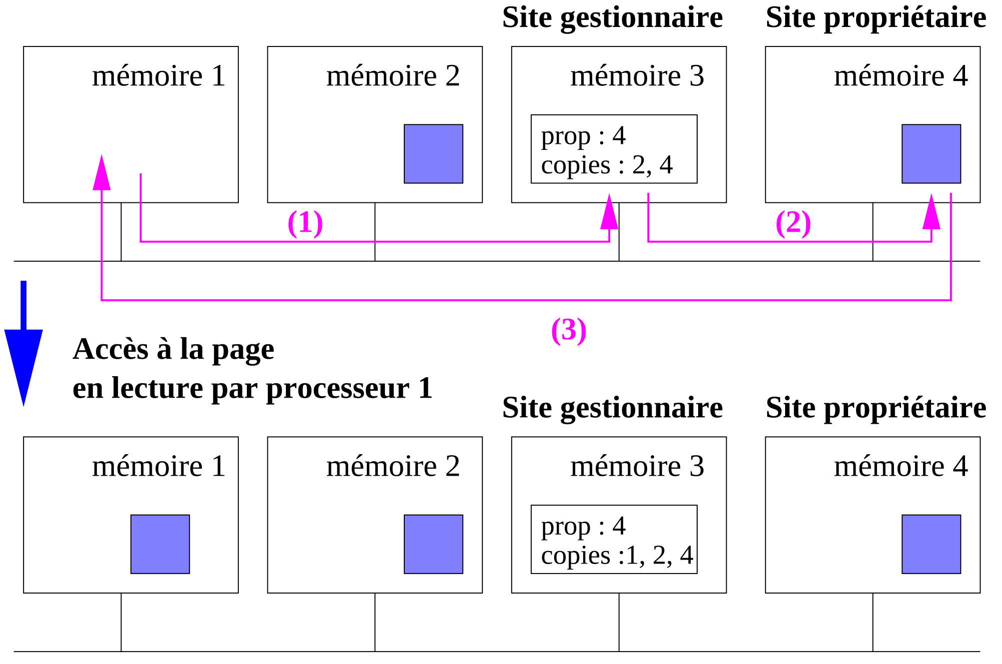
- *Propriétaire* : nœud ayant écrit en dernier sur la page
- *Gestionnaire* : nœud qui connait le propriétaire d'une page et qui est chargé de gérer les accès en écriture à la page
- *Ensemble de copies* : ensemble des nœuds possédant des copies de la page

Protocole de Li & Hudak (gestionnaire centralisé)

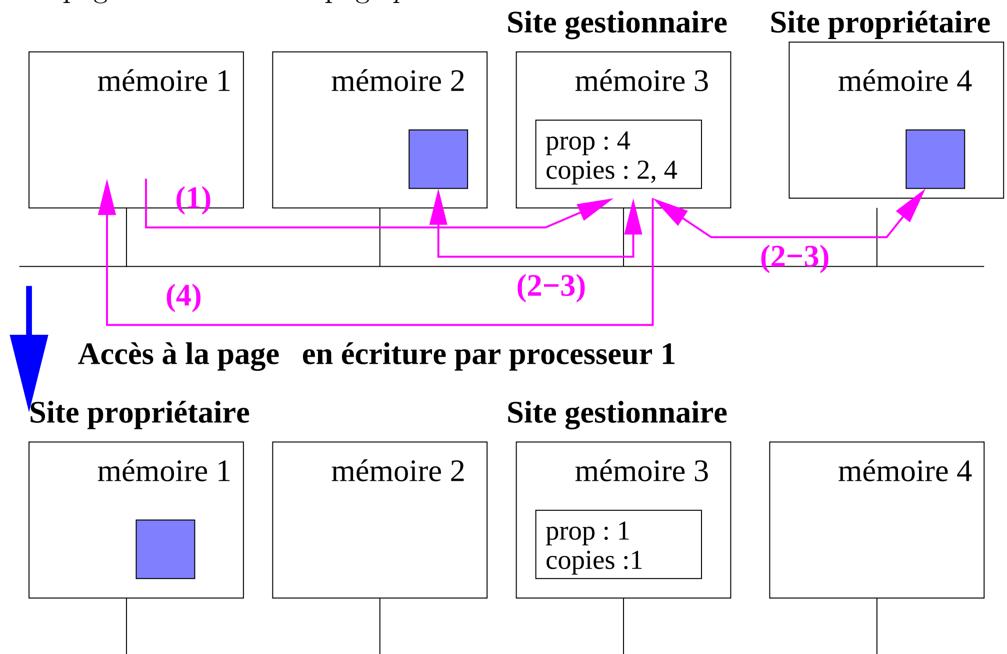
- *MRSW (Multiple Reader, Single Writer)* : n exemplaires en lecture ou un seul en écriture
- Modèle de *cohérence séquentielle*
- Protocole de cohérence à *invalidation sur écriture*
- Gestionnaire unique par page partagée

Défaut de page en lecture sur page p :

— Obtenir une copie de p auprès du gestionnaire de p, qui contacte le propriétaire
— Relancer l'instruction
Défaut de page en lecture sur page p :



Défaut de page en écriture sur page p :
— Obtenir une copie de p auprès du gestionnaire de p, qui contacte le propriétaire
— Invalider les autres copies de p
— Changement de propriétaire
— Relancer l'instruction
Défaut de page en écriture sur page p :

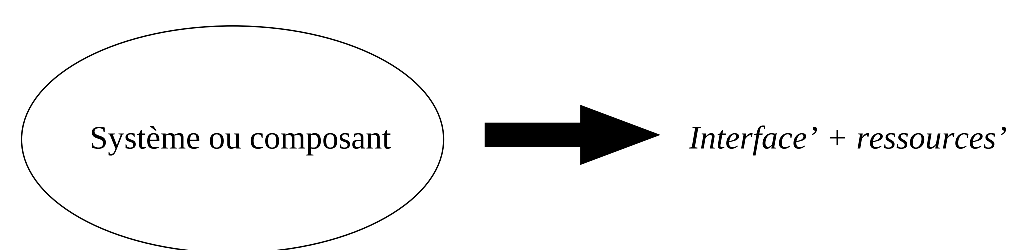


Septième partie

Virtualisation

1 Définition et intérêt

Virtualisation En général



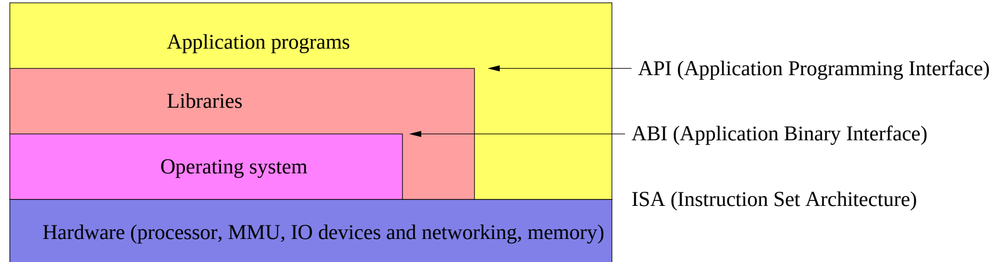
Virtualisation Exemples

- Mémoire virtuelle (virtualisation de la mémoire physique)
- Virtual Machines (virtualisation de la machine physique)
  - JVM
  - Autres jeux d'instructions virtualisés (CLI)
- Fichier (virtualisation du disque)
- Système d'exploitation (virtualisation de la machine physique)

$\Longrightarrow$ Un concept aussi vieux que l'informatique

Virtualisation Machines virtuelles

Principe de virtualisation appliquée à la machine entière



Classification des machines virtuelles(Smith & Nair, 2005) :

- *Process-based* : la machine virtuelle exécute un processus unique (abstraction au niveau ABI ou API). JVM, .NET
- *System-based* : fournit un support pour un système d'exploitation et son ensemble de processus (abstraction au niveau ISA) : Xen, KVM, VMware, VirtualBox

Virtualisation Intérêts

- Abstraction
- Ajout de fonctionalités
- Différents services sur une même machine avec partage des ressources physiques. Isolation
  - Spatiale
  - Temporelle
- Exécution d'applications sur des systèmes/architectures plus maintenues
- Facilité d'inspection et contrôle
- Tolérance aux fautes

— Sauvegarde/migration/re-démarrage de la machine virtuelle
— Pas de risque de crasher la machine

Objectifs de la présentation

- Faire cohabiter plusieurs systèmes d'exploitation sur la même machine
  - Utilisation de la virtualisation : abstraire la machine physique pour la partager entre plusieurs systèmes d'exploitation
  - Virtualisation system-based
  - Concept assez ancien : *Architecture of Virtual Machines, R.P. Goldberg, 1973*
- Différentes techniques de virtualisation
- Support matériel pour la virtualisation

2 Techniques de virtualisation

Principe général

- Machine virtuelle : couche logicielle émulant la plateforme (processeur + matériel), s'exécutant sur la machine hôte
- Système d'exploitation invité (guest) s'exécute au dessus de la machine (virtuelle) simulée

Techniques de virtualisation

Localisation machine virtuelle :

- Type I (*native, bare-metal*) : couche logicielle de virtualisation (*hyperviseur*) directement sur la machine hôte
- Type II : (*hosted*) couche logicielle de virtualisation au-dessus du système hôte

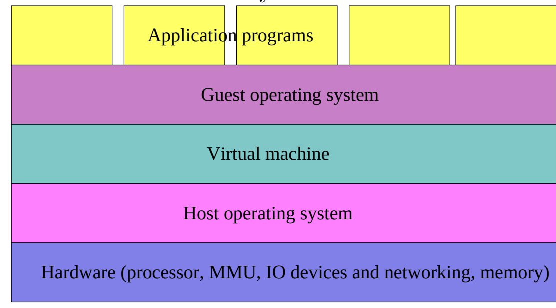
Conformité à la machine réelle :

- *Full-virtualization* : la couche de virtualisation de plateforme est suffisamment complète pour exécuter un système invité non modifié (souvent type II).
- *Para-virtualization* : couche de virtualisation adaptée pour de meilleures performances (type I).

2.1 Full-virtualization

Full virtualization

- Simulation complète d'une plateforme matérielle (processeur + périphériques)
- Couche de virtualisation au dessus du système hôte



Full virtualization

Principes de réalisation :

- Processeur de l'hôte ≠ processeur de l'invité : simulation par logiciel du jeu d'instruction de l'invité (lent)
- Processeur de l'hôte = processeur de l'invité : éviter la simulation des instructions standard (load/store, instructions arithmétiques et logiques), simuler l'exécution des instructions "non sûres" (opérations privilégiées)
  - VMWare : Transformation binaire à la volée du code x86 pour remplacer le code des instructions "non sûres" par une simulation à l'intérieur de la machine virtuelle
- Note : Sur x86, demande de placer un niveau de virtualisation sous l'OS (anneau de protection 0) pour exécuter les instructions privilégiées

Full virtualization : intérêts

- Permet d'émuler un processeur différent de celui du matériel hôte
- Systèmes d'exploitation hôte et invité ne sont pas modifiés
- Excellente isolation entre les machines virtuelles et le système hôte
- Très bonnes performances quand les processeurs de l'hôte et de l'invité sont identiques
- Possibilité de mémorisation de l'état de la machine virtuelle

Full virtualization : inconvénients

- Forte pénalité sur les performances lorsque les processeurs de l'hôte et de l'invité sont différents

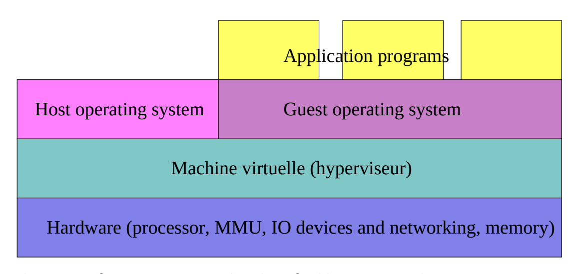
$\Longrightarrow$ Compilation dynamique de binaire à binaire (just-in-time)

Full virtualization : exemples

- VMWare
- Bochs
- Microsoft Virtual PC, VirtualServer
- VirtualBox
- Qemu

2.2 Paravirtualization

Paravirtualisation



- Objectif : améliorer la performance de la full-virtualization

Paravirtualisation

Hyperviseur :

- Noyau allégé et spécialisé dans la virtualisation

- Déportation dans l'hyperviseur les opérations trop lentes à effectuer dans l'environnement virtualisé (ex : gestion du temps, gestion d'interruptions, drivers)
- Hyperviseur tourne directement sur la machine hôte
- Les noyaux invités sont conscients d'être virtualisés et font appel à l'hyperviseur pour traiter les appels système

Paravirtualization : intérêts
- Performances supérieures à la full-virtualisation
- Très bonne isolation

Paravirtualization : inconvénients
- Nécessite de modifier le système invité
- L'architecture des systèmes hôte et modifié doivent être les mêmes

Paravirtualization : exemples
- Xen
- Denali

Full virtualization vs paravirtualisation
- Full virtualization : developpement, fiabilité
- Para virtualization : performance

2.3 Exemple de Qemu

Qemu : caractéristiques
- Full virtualization, type II
- Fonctionne de manière stable sur les architectures suivantes :
  - x86 (Linux, Windows et Mac OSX)
  - x86 64 (Linux)
  - PowerPC (Linux et Mac OSX)
- Emule de manière stable les plateformes suivantes :
  - x86
  - x86 64
  - ARM
  - PowerPC
  - Mips
  - Sparc
- Son code source a été repris dans différents autres projets (VirtualBox, KVM)

Qemu : compilation dynamique
- Chaque instruction de l'architecture invitée est décomposée en micro-instructions
- Micro-instruction : instruction ultra-simplifiée, mode d'adressage très simple
- Chaque micro-instruction est programmée (en C) comme une fonction capable d'effectuer le calcul en question
- Compilation de chacune des fonctions vers un fichier objet pour l'architecture hôte (gcc)

Qemu : compilation dynamique

— Traduction de l'instruction PowerPC suivante vers l'architecture x86 :

```
addi r1,r1,-16                  # r1 = r1 - 16
```

— Les micro-instructions associées sont :

```
movl_T0_r1                      # T0 = r1
addl_T0_im -16                  # T0 = T0 + (-16)
movl_r1_T0                      # r1 = T0
```

(T0 est une variable temporaire dont on précise à GCC qu'elle doit être stockée dans un registre du processeur hôte)

Qemu : compilation dynamique

Fonctions C correspondantes et code natif généré

```
void op_movl_T0_r1(void) {
    TO = env → regs[1];                    mov 0x4(%ebp),%ebx
}

extern int _op_param1;
void op_add_T0_im(void) {
    T0 = T0 + ((long) &_op_param1);        add $0xfffffff0,%ebx
}

void op_mov_r1_T0(void) {
    env → regs[1] = T0;                    mov %ebx,0x4(%ebp)
}
```

Qemu : compilation dynamique

— Le compilateur dynamique traduit les blocs de base au fur et à mesure de leur exécution.
— Les blocs déjà traduits sont conservés dans une table de hachage (cache)

2.4 Hardware-assisted virtualization

Hardware-assisted virtualization

— Intel Virtualization technology (VT-x) and AMD AMD-V (Pacifica)
— Niveau de protection supplémentaire pour la machine virtuelle, descripteurs pour état de la machine virtuelle
— Tables des pages de l'OS invité gérées par matériel et plus émulées
— Tagged TLB entries : VPIDs (Virtual Processor IDs) pour les machines virtuelles